Documentation de la plate−forme JOnAS

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Documentation JOnAS

Java Open Application Server (JOnAS): une Plate−forme J2EE™

Dernière modification le 14−11−2003,, JOnAS 3.3.1

Ce document fournit une vue d'ensemble de la plate−forme J2EE JOnAS. Son contenu est le suivant :

Introduction

J2EE

Les spécifications J2EE Sun™, ainsi que leurs spécifications relatives (EJB™, JMS™,...), définissent une architecture et des interfaces pour le développement et le déploiement d'applications internet distribuées, développées en Java™ et basées sur une architecture multi−tiers.

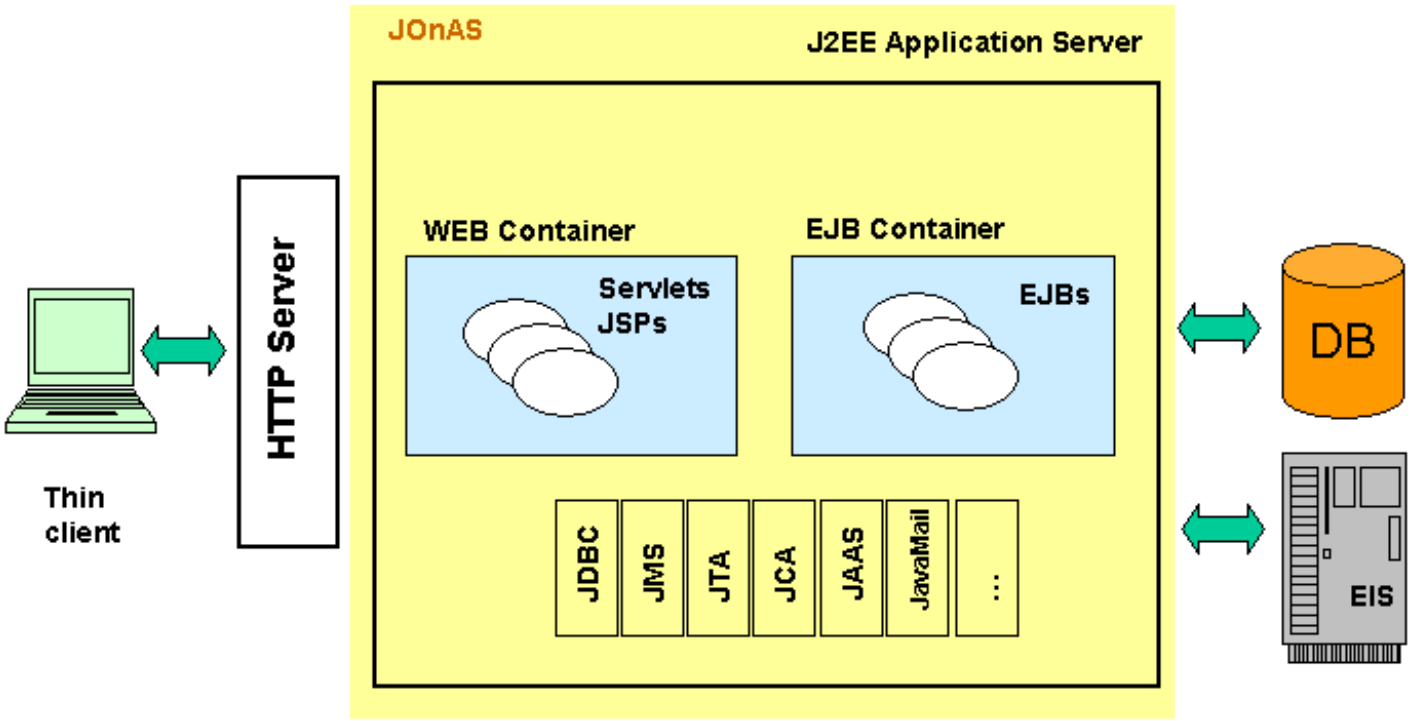
Ces spécifications ont pour but de faciliter et normaliser le développement, le déploiement, et l'intégration de composants applicatifs ; Ces composants pourront être déployés sur les plates−formes J2EE. Les applications résultantes sont typiquement des applications Web, des applications transactionnelles, orientées bases de données, à utilisateurs multiples, sécurisées, scalables, et portables. De façon plus précise, ces spécifications décrivent deux types d'informations :

- Le serveur J2EE en premier lieu, qui fournit l'environnement d'exécution et les services système requis, tels que le service transactionnel, la persistance, le service de messagerie Java (JMS), et le service de sécurité
- Des informations à destination du programmeur et de l'utilisateur, précisant les règles de développement, de déploiement et d'utilisation des composants applicatifs.

En suivant ces spécifications, un composant applicatif sera non seulement indépendant de la plate−forme et du système d'exploitation (puisqu'il est écrit en langage JAVA), mais également de la plate−forme J2EE.

Une application J2EE typique se compose 1) de composants de présentation, également appelé des composants Web (Servlets et JSPs™), qui définissent l'interface de l'application Web, et 2) de composants métiers, les Enterprise JavaBeans (EJB), qui définissent la logique métier de l'application et les données applicatives. Le serveur J2EE fournit des conteneurs pour héberger les composants Web et les composants métiers.

Le conteneur assure la gestion du cycle de vie des composants et interface les composants avec les services proposés par le serveur J2EE. Il existe deux types de conteneurs : le conteneur Web, qui prend en charge les composants Servlet et JSP, et le conteneur EJB qui prend en charge les Enterprise JavaBean. Un serveur J2EE peut également fournir un environnement de déploiement de clients Java (utilisant des EJBs), on parle alors de conteneur client.



ObjectWeb

JOnAS est une implémentation Open Source J2EE, développée dans le cadre du consortium ObjectWeb. ObjectWeb est une initiative open source, qui peut être comparée à Apache ou Linux, mais dans le domaine du "middleware" (intergiciel). La finalité d' ObjectWeb est de développer et de promouvoir des logiciels "middleware" open source.

ObjectWeb est un Consortium International piloté par l'Inria, fondé officiellement en 2002 par Bull, France Telecom et l'INRIA. Tout logiciel est diponible sous la licence LGPL.

L'objectif technique de ce consortium est de développer des intergiciels à base de composants distribués, en ligne avec les standards CORBA, Java et W3C. L'intention est d'appliquer le modèle composant, tel qu'il est déjà utilisé au niveau applicatif dans J2EE ou dans le modèle composant de CORBA, au niveau du "middleware" lui−même. La

couverture fonctionnelle d'ObjectWeb adresse le nommage, le "trading", la communication (par événements ou messages), la disponibilité et la fiabilité (gestion des transactions, persistence, réplication, tolérance aux fautes), la répartition de charge et la sécurité. D'autres domaines associés sont aussi traités dans le Consortium, tels que la robustesse, l'optimisation, la qualité du code, les "benchmarks", les tests, les démonstrateurs et les outils de développement.

Ainsi, le modèle global d'architecture ObjectWeb va des applications (telles que Rubis, benchmark issu du projet JMOB) jusqu'aux plate−formes "middleware" (JOnAS, OpenCCM ou ProActive) sur lesquelles elles tournent. Les plates−formes sont basées sur des composants techniques tels un Middleware−Orienté−Message (JORAM qui implémente JMS), un environnement de communication (CAROL), un environnement de gestion de la persistence (JORM), un environnement d'interrogation de base de données (MEDOR), un moniteur transactionnel (JOTM), un Object Request Broker (Jonathan). Un composant technique tel que C−JDBC permet à toute plate−forme de mettre en oeuvre des clusters de bases de données. JOnAS utilise tous ces composants ObjectWeb (JORAM, CAROL, Jonathan, JORM, MEDOR, C−JDBC, et bientôt JOTM), mais également des composants open source d'autres communautés, tels Tomcat ou Jetty, utilisés comme conteneur Web, ou AXIS, utilisé pour fournir l'environnement "Web Services".

ObjectWeb regroupe déjà un nombre significatif de membres corporate, des universités ainsi que des membres individuels (l'adhésion individuelle est gratuite). Les membres d'ObjectWeb contribuent aux orientations d'ObjectWeb et participent à tous les groupes de travail, réunions, ateliers, et conférences. La communauté des développeurs et des utilisateurs travaillant avec des composants et des plate−formes ObjectWeb se développe constamment.

Les fonctionnalités de JOnAS

JOnAS est un serveur d'application pur Java, open source, se conformant à la spécification J2EE. En outre sa forte modularité lui permet d'être utilisé comme :

- un serveur J2EE , pour déployer et exécuter des applications EAR (c. à. d. pour des applications composées simultanément de composants web et d'Ejb),
- un conteneur d' EJB , pour déployer et exécuter des composants EJB (c. à. d. pour des applications sans interfaces Web ou utilisant des moteurs de JSP/Servlet qui ne sont pas intégrés comme conteneur J2EE JOnAS),
- un conteneur Web, pour déployer ou exécuter des JSPs et Servlets (c. à. d. pour des applications sans composants EJB).

Pré−requis Système

JOnAS est disponible pour JDK version 1.3 or 1.4. Il a été utilisé sur plusieurs systèmes d'exploitation (Linux, AIX, Windows, Solaris, HP−UX, etc.), et avec différentes bases de données (Oracle, PostgreSQL, MySQL, SQL server, Access, DB2, Versant, Informix, Interbase, etc.).

Conformité aux standards Java

JOnAS est une implémentation des spécifications J2EE 1.3. et respecte actuellement les spécifications EJB 2.0. Son intégration de Tomcat ou Jetty en tant que conteneur Web lui assure la conformité aux spécifications Servlet 2.3 and

JSP 1.2 . Le serveur Jonas supporte ou implémente les API Java suivantes : JCA™ 1.0, JDBC™ 2.0, JTA™ 1.0.1, JMS™ 1.1, JMX™ 1.0, JNDI™ 1.2.1, JAAS™ 1.0, JavaMail™ 1.3.

Fonctions clés

Outre l''implémentation de tous les standards J2EE, JOnAS propose les fonctionalités avancées suivantes :

- ***Administration:*** L'administration du serveur JOnAS utilise JMX et fournit une console d'administration basée sur les JSP (Struts).
- ***Services:*** L'architecture de JOnAS, basée sur les services, assure au serveur un haut degré de modularité et de configurabilité. Ceci permet au développeur d'appliquer au niveau middleware une approche basée sur un modèle composant, et rend plus facile l'intégration des nouveaux modules (p. ex. pour les contributeurs opensource). Ceci permet également de démarrer uniquement les services nécessaires à chaque application, et d'économiser ainsi les ressources système. Les services JOnAS sont administrables avec JMX.
- ***Scalabilité:*** JOnAS intègre plusieurs mécanismes d'optimisation pour augmenter la scalabilité du serveur. Ceci comprend un pool de beans session, un pool de beans Message−Driven, un pool de threads, un cache de beans entités, l'activation/désactivation des beans entités, un pool de connexions (pour JDBC, JMS, J2EE CA), et l'optimisation des accès aux supports de stockage (shared flag, isModified).
- ***Mise en cluster:*** Les solutions de "Clustering" JOnAS proposent au niveau WEB aussi bien qu'EJB l'équilibrage de charge, la haute disponibilité et la résistance aux pannes.
- ***Distribution:*** JOnAS fonctionne avec différents environnement distribués, du fait de l'intégration du projet ObjectWeb CAROL (Common Architecture for RMI ObjectWeb Layer), qui assure la prise en compte simultanée de plusieurs protocoles de communication :
  - *RMI* utilisant le protocole propriétaire Sun JRMP
  - *RMI* sur IIOP
  - *CMI*, le protocole de distribution en mode cluster de JOnAS
  - *Jeremie*, la personnalité RMI de l' Object Request Broker Jonathan, du consortium Objectweb. Utilisé avec Jeremie, JOnAS bénéficie de façon transparente de l'optimisation des invocations RMI locales.
- ***Support des "Web Services:"*** Grace à l'intégration d'AXIS, JOnAS permet aux composants J2EE d'accéder aux "Web services" (i.e. d'être des clients "Web Services"), ainsi que d'être déployés en tant que "Web Services Endpoints" (i.e. de fournir des "Web Services"). Le déploiement des clients "Web Services" standards et des "Web Services Endpoints" est supporté, tel qu'il est spécifié en J2EE 1.4.
- ***Support de JDO:*** En intégrant l'implémentation ObjectWeb de JDO, SPEEDO, ainsi que le connecteur J2EE−CA associé, JOnAS permet aux composants J2EE d'utiliser des fonctions JDO.

Trois aspects critiques de J2EE ont été implémentés très tôt dans le serveur JOnAS :

- ***J2EE−CA:*** Les systèmes d'information de l'entreprise (EIS − Enterprise Information Systems) sont facilement accessibles aux applications JOnAS. En respectant l'architecture des connecteurs J2EE (J2EE−CA), JOnAS permet le déploiement de tout connecteur se conformant à J2EE CA. L'EIS est alors disponible depuis un composant applicatif J2EE. Par exemple, les mainframes Bull GCOS peuvent être accédés depuis JOnAS au travers de leur connecteur HooX . De plus en plus, les connecteurs vont devenir une façon standard d'introduire des drivers JDBC (et des implémentations JMS, avec J2EE 1.4) dans une plate−forme J2EE. JOnAS intègre un connecteur JDBC qui peut être utilisé en remplacement du service Bases de Données de JOnAS, et qui propose en plus la gestion d'un pool de requêtes JDBC pré−compilées (PreparedStatement).

- ***JMS:*** Les implémentations de JMS peuvent êre facilement introduites dans JOnAS. Elles s'exécutent sous forme de services JOnAS dans la même JVM (Java Virtual Machine) ou dans une JVM séparée. JOnAS propopose des fonctionnalités d'administration qui masquent les APIs d'administration propriétaires de JMS. Actuellement, plusieurs implémentations de JMS peuvent être utilisées : l'implémentation open source JORAM d'ObjectWeb, SwiftMQ ou Websphere MQ.
- ***JTA:*** La plate−forme JOnAS supporte les transactions distribuées impliquant des composants multiples, ainsi que des ressources transactionnelles. Le support des transactions JTA est fourni par un moniteur transactionnel développé sur une implémentation du service transactionnel de CORBA (OTS).

Les packages JOnAS

JOnAS est téléchargeable sous trois formes différentes :

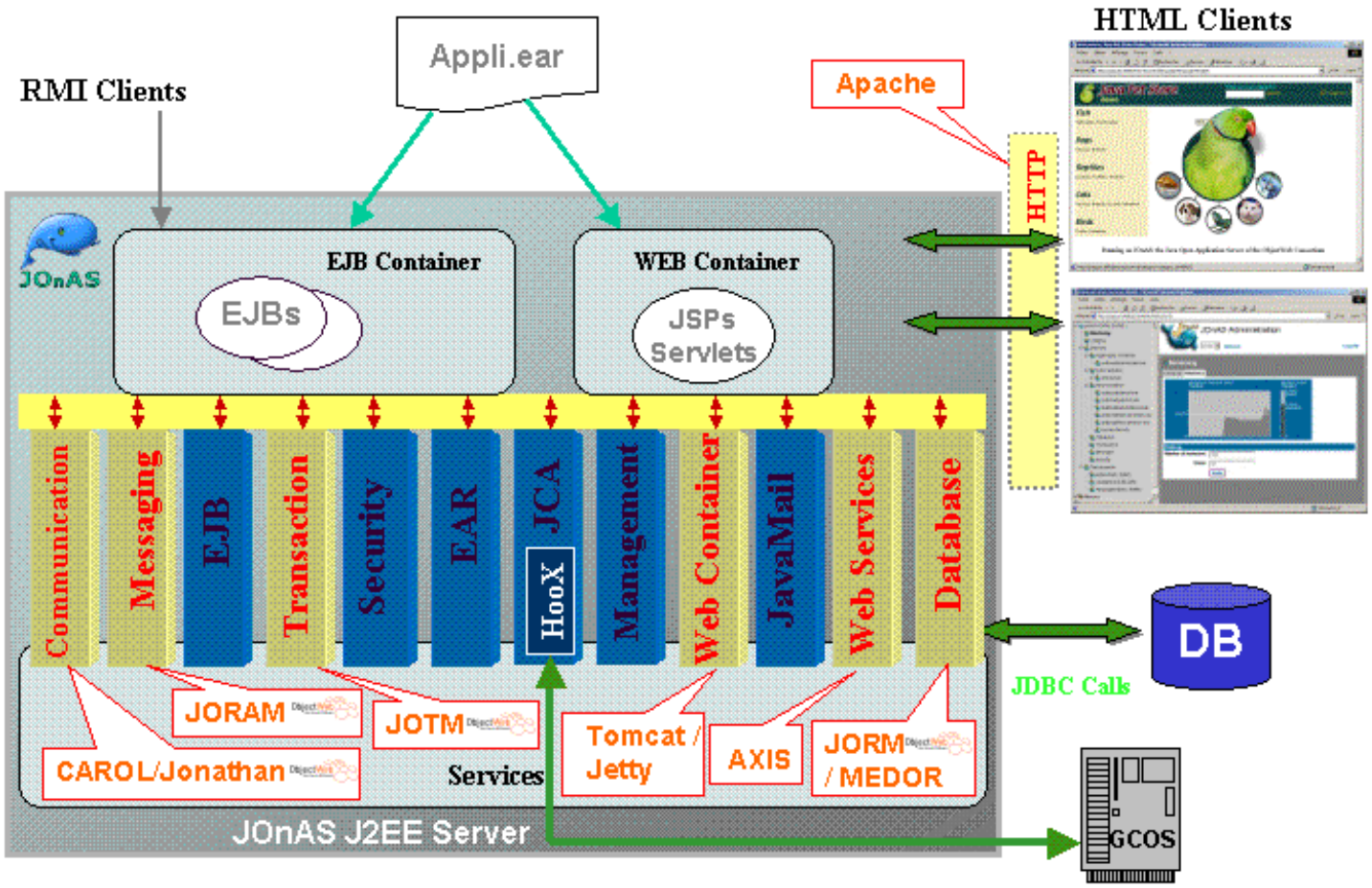
- Un packaging simple contient uniquement le serveur d'application, sans l'implémentation du containeur Web associé. Pour l'utiliser dans le cadre d'une application Web J2EE, Tomcat ou Jetty doivent être installés (dans une version adéquate) et doivent être configurés pour fonctionner avec JOnAS.
- Un package JOnAS−Tomcat contient à la fois JOnAS et Tomcat, pré−configurés, ainsi que des exemples pré−compilés. C'est un serveur d'application J2EE prêt à l'emploi.
- Un package JOnAS−Jetty contient à la fois JOnAS et Jetty, pré−configurés et avec des exemples compilés. C'est un serveur d'application J2EE prêt à l'emploi.

Ces packages intègrent également AXIS, qui fournit un support pré−configuré aux "Web−Services".

L'architecture JOnAS

L'architecture de JOnAS repose sur la notion de service. Typiquement, l'objectif d'un service est de fournir des ressources systèmes aux différents conteneurs. La plupart des composants du serveur d'application JOnAS sont des services JOnAS pré−définis. Cependant, il reste possible et facile pour un utilisateur JOnAS expérimenté de définir un nouveau service et de l'intégrer à JOnAS. Une application J2EE ne nécessite pas toujours l'intégralité des services disponibles, il est donc possible de définir lors de la configuration du serveur JOnAS, l'ensemble des services qui devront être chargés pendant la phase de lancement.

L'architecture JOnAS est illustrée par la figure suivante, présentant les conteneurs WEB et EJB reposant sur les services JOnAS (tous les services sont présents dans cette figure). Deux clients légers sont également présentés, l'un d'entre eux étant la console d'administration JOnAS (appelée JonasAdmin).



Service Communication et Service de Nommage

Ce service (appelé également "Registre") est utilisé pour charger le registre RMI, le CosNaming (Service de nommage CORBA), le registre CMI, et/ou le registre Jeremie, suivant la configuration JOnAS (configuration CAROL, qui spécifie les protocoles de communication à utiliser). Il existe différents modes de chargement des registres : dans la même JVM ou non, chargement automatique s'ils ne sont pas déjà en cours d'exécution. CAROL assure le support multi−protocole aussi bien à l'exécution qu'au déploiement, ce qui permet d'éviter de redéployer des composants lors d'un changement de protocole de communication.

Ce service fournit l'API JNDI aux composants applicatifs et aux autres services, pour leur permettre d'enregistrer et de retrouver ("bind" et "lookup") des objets distants (p. ex. "Home" des EJB) et des références aux ressources (JDBC "DataSource", "connection factories" Mail et JMS, etc.).

Service Conteneur EJB

Ce service a pour rôle le chargement des composants EJB et de leur conteneurs. Les conteneurs EJB sont constitués d'un ensemble de classes Java qui implémentent la spécification EJB, et d'un ensemble de classes d'interposition qui interfacent les composants EJB avec les services fournis par le serveur d'application JOnAS. Ces classes d'interposition sont spécifiques à chaque composant EJB et sont générées par l'outil de déploiement GenIC.

La configuration JOnAS permet de spécifier si ce service doit être lancé lors de l'initialisation de JOnAS.

Les Enterprise JavaBeans (EJB) sont des composants logiciels qui implémentent la logique métier de l'application (alors que les Servlets et JSPs en implémentent la présentation). Il existe trois types d'Enterprise JavaBeans :

- ***Les beans Session*** sont des objets associés à un client unique, d'une durée de vie limitée (un appel méthode, ou une session client), et qui représentent l'état courant de la session client. Ils sont avec ou sans état, et peuvent démarrer et/ou propager des transactions.
- ***Les beans Entité*** sont des objets représentant des données en base de donnée. Ils peuvent être partagés entre différents clients et sont indentifiés au moyen d'une clé primaire. Le conteneur EJB assure la gestion de la persistance de ce type d'objets. Cette gestion de la persistance est totalement transparente au client qui utilise l'objet, et peut être ou non transparente au développeur. Ceci dépend du type de persistence associé au bean entité :
  - Persistance gérée par le conteneur ***(Container−Managed Persistence)*** : dans ce cas, le programmeur ne développe aucun code d'accès aux données ; la gestion de la persistance est déléguée au conteneur. La correspondance entre le bean et le support de persistance est fournie dans le descripteur de déploiement, de façon spécifique au serveur d'application.
  - ***Persistence gérée par le bean (Bean−Managed Persistence)*** : Dans ce cas, le programmeur écrit toutes les opérations d'accès à la base de donnée dans les méthodes de l'EJB relatives aux opérations de création de données, de stockage, de chargement, et de destruction.
- ***Les beans Message−driven*** peuvent être considérés comme des objets à l'écoute de messages. Ils s'exécutent sur réception d'un message JMS (Java Message Service) ; ils sont sans état et peuvent déclencher et/ou propager des transactions. Ils implémentent une façon asynchrone d'invoquer des méthodes d'EJB.

La configuration JOnAS permet de définir un ensemble de fichiers ejb−jar à charger par le conteneur. Les fichiers ejb−jar peuvent également être chargés en cours d'exécution du serveur.

Pour implémenter la persistance gérée par le conteneur conformément aux spécifications EJB 2.0 (Container−Managed − CMP2), JOnAS s'appuie sur les environnements d'ObjectWeb JORM (Java Object Repository Mapping) et MEDOR (Middleware Enabling Distributed Object Requests). JORM supporte la correspondance complexe d' EJB vers des tables en base de données, ainsi que différents types de stockage (bases de données relationnelles, bases de données objets, annuaires LDAP, etc ...).

JOnAS implémente déjà les fonctionnalités de "Timer Service" conformément aux spécifications EJB 2.1 (J2EE 1.4).

Service Conteneur Web

Le rôle de ce service est de lancer le moteur de Servlet/JSP dans la JVM du serveur JOnAS, et de charger les applications web (fichiers war) dans ce moteur.

Actuellement, ce service peut être configuré pour utiliser Tomcat ou Jetty. Les moteurs de Servlets/JSP sont intégrés à JOnAS en tant que "Conteneur Web", c'est à dire que de tels conteneurs fournissent aux composants web les accès aux ressources systèmes du serveur d'application et aux composants EJB, conformément aux spécifications J2EE.

La configuration de JOnAS permet de spécifier si ce service doit être lancé pendant la phase d'initialisation de JOnAS. Elle permet également de définir un ensemble de fichiers war à charger. Les fichiers war peuvent également être déployés en cours d'exécution du serveur, en utilisant les outils d'administration de JOnAS. La gestion des utilisateurs a été unifiée pour Tomcat/Jetty et JOnAS. Il est possible de configurer la politique de délégation de chargement de classe (priorité au chargeur de classe Webapp, ou au chargeur de la classe parent).

Les Servlet et JSP™ constituent des technologies utilisées pour développer des pages web dynamiques. L'approche Servlet permet le développement de classes Java (HTTP Servlets) pouvant être invoquées au travers de requêtes HTTP et générant des pages HTML. Typiquement, les Servlets accèdent au système d'information en utilisant les APIs JAVA (JDBC par exemple, ou les APIs des composants EJB) pour élaborer le contenu de la page HTML à générer en réponse à la requête HTTP. La technologie JSP est un complément à la technologie Servlet. Une JSP est une page HTML contenant du code Java à l'intérieur de tags particuliers de type XML ; ce code Java est chargé de générer dynamiquement le contenu de la page HTML.

Les Servlets et JSPs sont considérés comme des composants applicatifs J2EE, responsables de la logique de présentation de l'application. De tels composants applicatifs peuvent accéder aux ressources fournies par le serveur J2EE (comme les sources de données JDBC, les "connection factories" JMS , les EJBs, les "factories" Mail). Pour les composants J2EE, l'affectation effective de ces ressources (avec la ressource réellement mise à disposition par le serveur) s'effectue en fait lors du déploiement du composant, et est spécifiée dans le descripteur de déploiement de chaque composant, dès lors que son code utilise des noms logiques pour les ressources.

Service Ear

Ce service est utilisé pour des applications J2EE complètes, c.à.d. des applications packagées dans des fichiers EAR, contenant eux−mêmes des fichiers ejb−jar ou war. Ce service prend en charge les fichiers EAR, délègue le déploiement des fichiers war au Service Conteneur WEB, et celui des fichiers ejb−jar au service Conteneur EJB. Il prend en charge la création des chargeurs de classe appropriés de façon à ce que l'application J2EE s'exécute proprement.

Pour le déploiement d'application J2EE, JOnAS doit être configuré pour lancer le service EAR, et en spécifiant l'ensemble des fichiers EAR à charger. Les fichiers EAR peuvent également être déployés en cours d'exécution du serveur, en utilisant les outils d'administration de JOnAS

Service Transaction

Ce service obligatoire est fourni par le moniteur transactionnel JOnAS JTM, qui gère les transactions distribuées. Il gère les transactions pour les composants EJB de la façon dont elles sont définies dans leurs descripteurs de déploiement. Il gère la validation à 2 phases, face à un nombre quelconque de gestionnaires de ressources (Ressources XA). Dans la spécification J2EE, une ressource transactionnelle peut être une connexion JDBC, une session JMS, ou une connexion d'un connecteur J2EE CA. Le contexte transactionnel est propagé de façon implicite avec les requêtes distribuées. Le moniteur JTM peut être distribué sur un ou plusieurs serveurs JOnAS ; de ce fait, une transaction peut concerner différents composants localisés sur différents serveurs JOnAS. Ce service implémente la spécification JTA 1.0, et permet ainsi aux composants applicatifs ou aux applications clientes de démarrer ou de stopper les transactions de façon explicite. Seuls les composants applicatifs de type composants Web, les beans Session ou les beans "Message−driven" peuvent démarrer explicitement une transaction dans leur code. Dans ces deux derniers cas (bean session ou bean "Message−driven"), on parle de "Bean−managed transaction demarcation".

Un des avantages principaux du support des transactions par les EJB est cet aspect déclaratif, qui signifie que le contrôle des transactions n'est plus codé en dur dans l'application, mais configuré lors du déploiement. Ces aspect est connu sous le terme "*Container−managed transaction demarcation*." Avec le "*Container−managed transaction demarcation*," le comportement transactionnel d'un Enterprise Bean est défini lors de la configuration et fait partie du descripteur de déploiement du bean.

Suivant la valeur des attributs transactionnels associés aux méthodes EJB dans le descripteur de déploiement, le conteneur d'EJB fournit la démarcation de la transaction pour l'Enterprise Bean. Les attributs peuvent prendre une des valeurs suivantes :

- ***`NotSupported:`*** Si la méthode est invoquée au sein d'une transaction, la transaction est suspendue pendant le temps d'exécution de la méthode.
- ***`Required:`*** Si la méthode est invoquée au sein d'une transaction, alors la méthode est exécutée dans le contexte de cette transaction. Sinon, une nouvelle transaction est démarrée pour l'exécution de la méthode, et est validée avant que le résultat de la méthode ne soit envoyé à l'utilisateur.
- ***`RequiresNew:`*** La méthode sera toujours exécutée dans le contexte d'une nouvelle transaction. La nouvelle transaction est démarrée pour l'exécution de la méthode, et est validée avant que le résultat de la méthode ne soit envoyé à l'utilisateur. Si la méthode est invoquée au sein d'une transaction, celle−ci est suspendue avant que la nouvelle transaction ne soit démarrée, et restaurée après la fin d'exécution de cette dernière.
- ***`Mandatory:`*** La méthode devra toujours être invoquée dans le contexte d'une transaction, sinon le conteneur va faire remonter l'exception `TransactionRequired` .
- ***`Supports:`*** La méthode devra toujours être invoquée dans le contexte transactionnel de l'appelant. Si l'appelant n'a pas de transaction associée, la méthode est invoquée en mode non transactionnel.
- ***`Never:`*** Avec cet attribut, le client doit appeler la méthode en mode non transactionnel, sinon le conteneur remonte l'erreur `java.rmi.RemoteException`.

ObjectWeb a créé le projet JOTM (Java Open Transaction Manager), basé sur le service de transaction JOnAS JTM. Ce service transactionnel a été enrichi pour proposer des fonctionnalités transactionnelles avancées, telles les transactions imbriquées (nested transactions) et les transactions "Web Services" (une implémentation de DBTP est disponible). JOTM sera intégré à JOnAS très prochainement.

Service Base de données (Database)

Ce service a en charge la gestion des objets sources de données "Datasource". Un objet Datasource est un objet administratif standard JDBC utilisé pour gérer les connexions à une base de données. Le service Base de Données crée et charge ce type d'objets dans le serveur JOnAS. Les Datasources à créer et déployer peuvent être spécifiés lors de la configuration de JOnAS, ou après le lancement du serveur en utilisant les outil d'administration de JOnAS. Le service Base de Données a également en charge la gestion du pool de connexions aux bases de données pour les composants applicatifs, afin d'éviter un surnombre de créations de connexions physiques, qui sont coûteuses en temps. Le service Base de données peut maintenant être remplacé par le connecteur JDBC, à déployer par le Service Ressource J2EE−CA. Celui−ci fournit en outre un pool de requêtes JDBC pré−compilées (JDBC PreparedStatement pooling).

Service Sécurité

Ce service implémente les mécanismes d'autorisation pour accéder aux composants EJB, conformément aux spécifications EJB. La sécurité des EJB est basée sur le concept des ***rôles***. Les méthodes peuvent être accédées par un ensemble défini de rôles. Pour pouvoir invoquer une méthode, un utilisateur doit avoir accès à au moins un des rôles de cet ensemble. L'adéquation entre les rôles et les méthodes (en ce qui concerne les permissions) est faite dans le descripteur de déploiement, en utilisant les éléments `security−role` et `method−permission`. Il est également possible de programmer la gestion de la sécurité en utilisant deux méthodes de l'interface EJBContext, dans le but de renforcer ou de compléter la vérification de la sécurité dans le code du bean : `getCallerPrincipal()` et `isCallerInRole (String roleName)`. Les noms de rôles utilisés dans le code de l'EJB (par exemple dans la méthode isCallerInRole) sont en fait des références aux rôles de sécurité définis par ailleurs, ce qui rend le code de l'EJB indépendant de la configuration de la sécurité décrite dans le descripteur de déploiement. Le programmeur met ces références de rôles à disposition du déployeur de composant ou du concepteur d'application par le biais des éléments `security−role−ref` inclus dans les éléments `session` ou `entity` du descripteur de déploiement.

Avec JOnAS, la correspondance entre les rôles et l'identification des utilisateurs est faite dans la base d'identification des utilisateurs. Cette base d'identification peut être stockée soit dans des fichiers, soit dans un annuaire JNDI (comme LDAP), soit dans une base de donnée relationnelle. Cette fonctionnalité est réalisée par l'implémentation des Realms correspondants pour Tomcat et Jetty (c. à. d. UserDatabaseRealm, MemoryRealm, JDBCRealm, JNDIRealm) et via les modules de login JAAS pour les clients lourds. Les Realm ont pour rôle de propager le contexte de sécurité au conteneur d'EJB pendant les appels aux EJB. Des modules de login JAAS sont fournis pour l'authentification des conteneurs Web et des clients Java. Une authentification basée sur les certificats est également disponible, avec le module de login CRLLoginModule pour la révocation des certificats.

Service Messagerie (JMS)

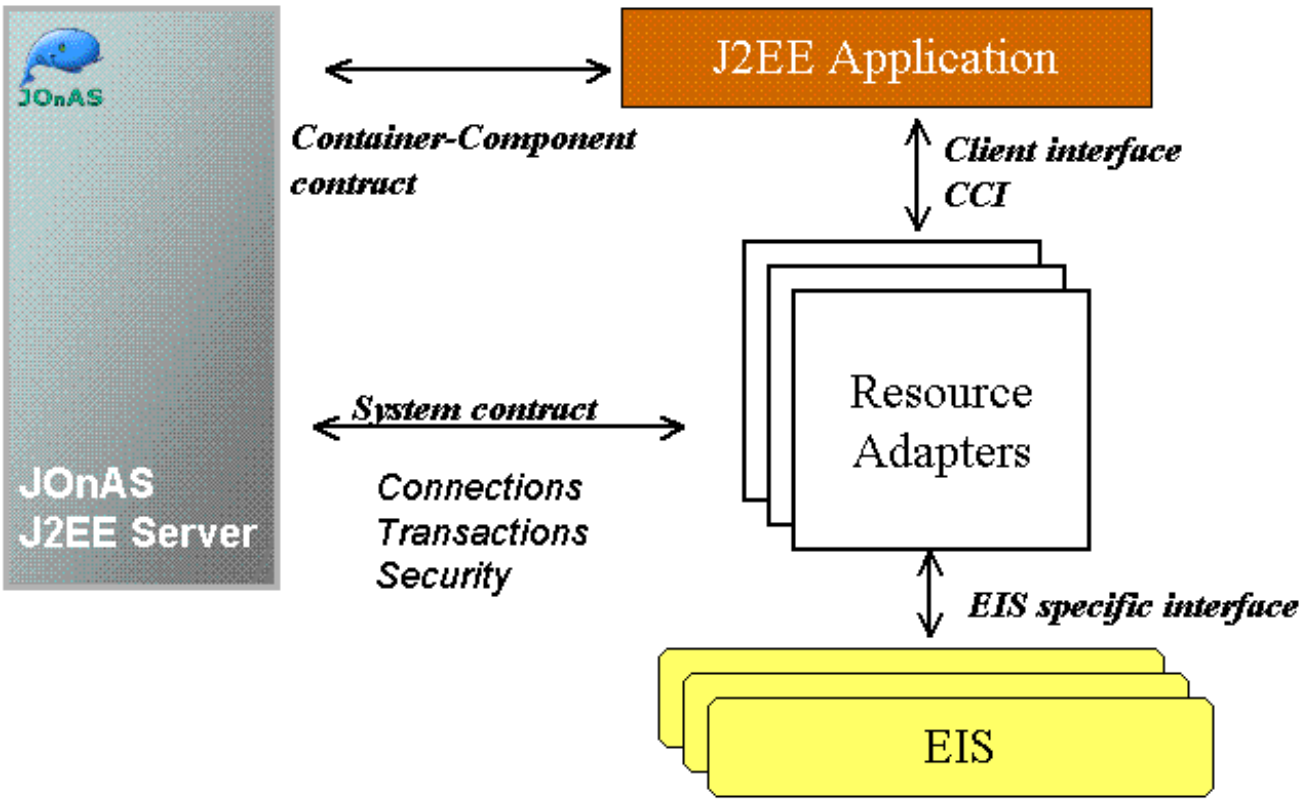
L'invocation asynchrone des méthodes EJB est possible avec les composants de type beans "Message−driven", conformément aux spécifications EJB 2.0. Un bean "Message−driven" est un composant EJB à l'écoute de messages, qui traite les messages JMS de façon asynchrone et qui peut donc être considéré comme un "JMS (Java Message Service) MessageListener". On lui associe une destination JMS, et sa méthode *onMessage* est activée sur réception d'un message envoyé par une application vers cette destination. Il est également possible à tout composant EJB d'utiliser l'API JMS pour des opérations JMS, qui s'effectueront de fait dans le contexte transactionnel géré par le serveur d'application.

Pour le support des beans "Message−driven" et des opérations JMS codées dans les composants applicatifs, le serveur d'application JOnAS repose sur des implémentations JMS développés par des tierces parties. Actuellement le logiciel open source JORAM est intégré et diffusé avec JOnAS, les produits SwiftMQ et WebSphere MQ peuvent également être utilisés, différentes implémentations JMS peuvent facilement être intégrées. JORAM propose un grand nombre de fonctionnalités intéressantes telles la fiabilité (en mode persistant), la distribution (de façon transparente au client JMS, il peut s'exécuter sur plusieurs serveurs, ce qui permet l'équilibrage de charge), et le choix entre TCP ou SOAP comme protocole de communication pour la transmission de messages.

Le service JMS a en charge le lancement du (ou l'établissement de la connexion au) serveur JMS intégré, qui peut ou non utiliser la même JVM que JOnAS. Il procure également un pool de connexions et de threads (pour les beans "Message−driven"). A travers ce service, JOnAS fournit des facilités pour créer des objets administrés JMS, tels les "connection factories" et les destinations. Ces objets peuvent être créés lors du lancement du serveur d'application, ou ultérieurement lors de son exécution en utilisant les outils d'administration du JOnAS.

Service Ressource J2EE−CA

L'architecture des connecteurs J2EE (Connector Architecture J2EE−CA) permet la connexion de différents Systèmes d'Information de l'Entreprise (EIS) à un serveur d'application J2EE. Cette fonctionnalité est basée sur le connecteur (Resource Adapter : RA), un composant d'architecture comparable à un driver logiciel, qui connecte l'EIS, le serveur d'application, et l'application (composants J2EE). Le connecteur (RA) est généralement fourni par le fournisseur de l'EIS, et offre aux composants J2EE une interface Java (CCI, pour Common Client Interface) pour accéder à l'EIS (il peut s'agir également d'une interface spécifique). Le connecteur fournit également une interface standard pour se brancher sur le serveur d'application, qui leur permet de collaborer pour garder tous les mécanismes systèmes (transactions, sécurité, et gestion des connexions) transparents aux composants applicatifs.



L'application exécute des opérations ayant trait à la "logique métier" sur les données de l'EIS en utilisant l'API client RA (CCI), alors que les transactions, connexions (incluant le pooling), et la sécurité sont gérées par le serveur d'application au travers du connecteur (system contract).

Le Service Ressource de JOnAS a en charge le déploiement sur le serveur JOnAS des connecteurs conformes aux spécifications J2EE−CA, qui sont packagés sous forme de fichiers RAR. Les fichiers RAR peuvent également être inclus dans des fichiers EAR. Dans ce cas, le connecteur sera chargé par le chargeur de classe de l'application. Une fois les connecteurs déployés, une instance de "connection factory" est disponible dans l'espace de nommage JNDI, et accessible aux composants applicatifs.

Un connecteur J2EE CA 1.0 pour JDBC est disponible avec JOnAS. Il peut remplacer le service de base de donnée actuel de JOnAS pour intégrer les drivers JDBC et gérer les pools de connexions. Il permet également de gérer des

pools de requêtes JDBC pré−compilées.

Service Administration JMX

Le service Administration est nécessaire pour administrer un serveur JOnAS depuis la console d'administration. Chaque serveur utilisant ce service est visible depuis cette console JOnAS. Ce service est basé sur JMX. Des MBeans standards (pour Management Beans) sont définis dans le serveur d'application JOnAS ; ils présentent les méthodes d'administration des objets instrumentés du serveur JOnAS, tels les services, les conteneurs, et le serveur lui−même. Le service Administration lance un serveur JMX (actuellement le serveur JMX Sun RI).

Les MBeans du serveur JOnAS sont enregistrés dans ce serveur JMX. La console d'administration JOnAS est une application Web basée sur Struts (à base de Servlet/JSP). Elle accède au serveur JMX pour présenter les fonctionnalités administrées. Ainsi, à l'aide d'un simple navigateur Web, il est possible d'administrer un ou plusieurs serveurs d'application JOnAS. La console d'administration permet de configurer tous les services JOnAS (et de rendre cette configuration persistante), de déployer tout type d'application (EJB−JAR, WAR, EAR) et tout type de ressource (DataSources, JMS et Mail Connection factories, Connecteurs J2EE−CA), tout ceci sans avoir à arrêter puis relancer le serveur. La console d'administration affiche les informations utiles pour piloter le serveur et les applications : utilisation mémoire, threads utilisées, nombre d'instances EJB, quel composant utilise quelle ressource, etc.. Lorsque le conteneur Web utilisé est Tomcat, il peut être administré à l'aide de la console JOnAS.

Service Mail

Un composant applicatif J2EE peut envoyer des messages par e−mail en utilisantJavaMail™. Le service mail du serveur d'application JOnAS fournit les ressources nécessaires à ces composants. Le service mail crée des "factories" Mail et enregistre ces ressources dans l'espace de nommage JNDI de la même manière que le service base de donnée ou le service JMS créent des "Datasources" ou des "ConnectionFactories" et enregistrent ces objets dans l'espace de nommage JNDI. Il y a deux types de "factories" Mail : *javax.mail.Session* et *javax.mail.internet.MimePartDataSource*.

Service WebServices

Ce service est implémenté au dessus d'AXIS, et est utilisé lors du déploiement de Web Services.

Environnement de développement et de déploiement sur JOnAS

Configuration de JOnAS et déploiement

Une fois JOnAS installé dans un répertoire référencé par la variable d'environnement JONAS_ROOT, il est possible de configurer des serveurs et de déployer des applications dans divers environnements d'exécution. Ceci se fait par le positionnement de la variable d'environnement JONAS_BASE. JONAS_ROOT et JONAS_BASE peuvent se comparer aux variables CATALINA_HOME et CATALINA_BASE de Tomcat. JONAS_ROOT pointe vers les répertoires d'installation de JOnAS, alors que JONAS_BASE spécifie une instance particulière de configuration du serveur d'application JOnAS. JONAS_BASE désigne le répertoire contenant une configuration particulière de JOnAS, ainsi que les sous−répertoires contenant les fichiers EJB−JAR, WAR, EAR et RAR qui peuvent être chargés dans cet

environnement d'application. Le fichier JOnAS build.xml comprend une balise ANT utilisée pour créer une nouvelle arborescence de répertoire JONAS_BASE. Ainsi à partir d'une installation unique de JOnAS, il est possible de basculer d'un environnement applicatif vers un autre en changeant uniquement le contenu de la variable JONAS_BASE. Il existe deux façons de configurer un serveur d'applications JOnAS et de charger des applications : soit en utilisant la console d'administration, soit en éditant les fichiers de configuration. Pour chaque type d'application ou de ressource (EJB−JAR, WAR, EAR, RAR) il existe également des répertoires "autoload", permettant un chargement automatique des applications et ressources localisées dans ces répertoires au lancement du serveur JOnAS.

JOnAS fournit différents outils pour le déploiement :

- Pour écrire les descripteurs de déploiement, certains plugins d'Environnements Intégrés de Développement (Integrated Development Environments : IDE) fournissent des fonctionnalités d'édition et de génération de code (les plugins Eclipse et JBuilder sont disponibles). L'outil NewBean intégré à JOnAS génère des modèles de descripteurs de déploiement. L'outil Xdoclet en génère également. Le projet ObjectWeb Apollon génère des interfaces graphiques utilisateurs pour éditer tout type de fichier XML ; il a été utilisé pour générer une interface graphique d'édition de descripteur de déploiement. Un outil de déploiement développé par la communauté ObjectWeb JOnAS (earsetup) sera également disponible pour travailler avec les API de déploiement JSR88 (J2EE 1.4) fournies par le projet ObjectWeb Ishmael .
- Il existe également des outils plus basiques pour le déploiement en lui même : la commande JOnAS GenIC ou sa tâche ANT associée (ejbjar). Les plugins IDE font appel à ces outils pour les opérations de déploiement. La fonctionnalité principale du projet Ishmael sera le déploiement d'applications sur la plate−forme JOnAS.

Environnements de développement JOnAS

Il existe plusieurs plugins et outils facilitant le développement d'applications J2EE devant être déployées sur JOnAS. Les plugins IDE pour JBuilder (Kelly) et Eclipse (JOPE) offrent la possibilité de développer, déployer et débugger des composants J2EE sur JOnAS. Le moteur Xdoclet, générateur de code, peut générer des interfaces EJB et des descripteurs de déploiement (standards, ou spécifiques à JOnAS), prenant en entrée la classe d'implémentation des EJB contenant des tags JavaDoc spécifiques. L'outil JOnAS NewBean génère des modèles d'interface, de classe d'implémentation et de descripteurs de déploiement pour tout type d'EJB. Beaucoup d'outils de développement peuvent fonctionner avec JOnAS, pour plus de détails, se référerer au document suivant : JOnAS tools page.

De plus, JOnAS est livré avec des exemples J2EE complets, fournissant le fichier ANT build.xml incluant toutes les balises nécessaires pour compiler, déployer et installer des applications J2EE.

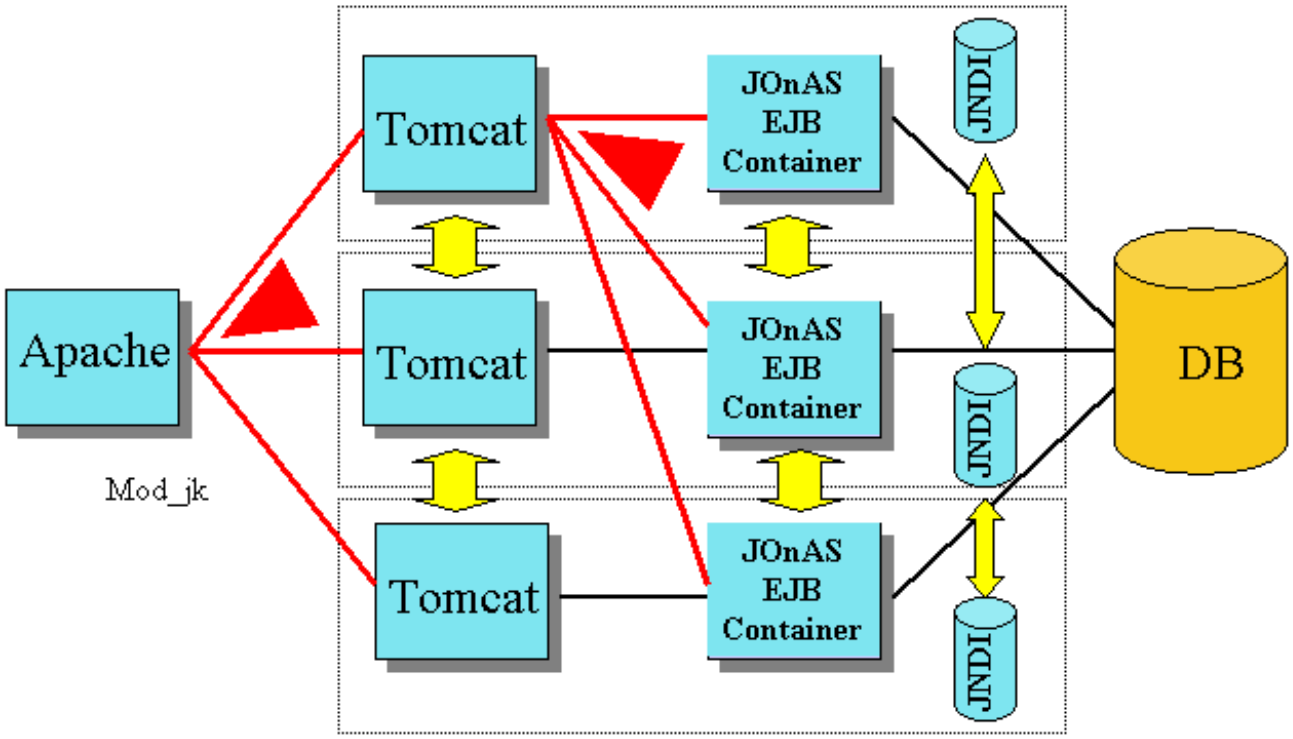
Mise en Cluster et Performance

La mise en cluster pour un serveur d'application couvre habituellement trois aspects : l'équilibrage de charge (Load Balancing − LB), la haute disponibilité (High Availability − HA) et la résistance aux pannes. De tel mécanismes peuvent être proposés au niveau du conteneur Web en répartissant les requêtes HTTP entre plusieurs instances de moteurs Servlet/JSP, au niveau du conteneur d'EJB en répartissant les requêtes EJB entre différentes instances de conteneur d'EJB, et au niveau base de données en utilisant plusieurs bases. La duplication du service de nommage JNDI est également nécessaire.

JOnAS assure l'équilibrage de charge, la Haute Disponibilité, et la résistance aux pannes au niveau du conteneur Web en utilisant le plugin Apache Tomcat mod_jk, et un mécanisme de réplication de session HTTP en mémoire basé sur JavaGroup. Le plugin distribue les requêtes HTTP depuis le serveur web Apache vers les instances de Tomcat vues comme des conteneurs Web. Les fluctuations du serveur sont automatiquement prises en compte. Ce plugin supporte les algorithmes de répartition de charge "round−robin et weighted round−robin", avec une option de "sticky session".

L'équilibrage de charge et la Haute Disponibilité sont aussi fournies dans JOnAS au niveau du conteneur EJB. Les opérations invoquées depuis les interfaces Home des EJB (création et recherche d'instances) sont réparties sur les noeuds du cluster. Le mécanisme est basé sur un registre JNDI répliqué, et en utilisant le protocole d'invocation de méthode à distance en mode cluster (Clustered remote Method Invocation protocol − CMI). Les stubs contiennent la connaissance du cluster et implémentent la politique d'équilibrage de charge ("round−robin" et "weighted round−robin"). Dans un futur proche, un mécanisme basé sur la charge de chacun des noeuds sera disponible. La résistance aux pannes au niveau EJB sera proposée en implémentant un mécanisme de réplication de l'état des bean session avec état.

L'architecture JOnAS en cluster est illustrée sur la figure suivante :



Apache est utilisé comme serveur HTTP en "front−end" ; Tomcat est utilisé comme conteneur Web. Les serveurs JOnAS partagent la même base de données. le plugin mod_jk fournit l'équilibrage de charge et la Haute Disponibilité au niveau Servlet/JSP. La résistance aux pannes est assurée au travers du mécanisme de réplication de session en mémoire. L'équilibrage de charge et la Haute Disponibilité sont fournies au niveau EJB par le protocole CMI, avec réplication du registre JNDI. Tomcat peut tourner ou non dans la même JVM que le conteneur EJB. JOnAS fournit un ensemble de documentations pour configurer une telle architecture.

L'utilisation du projet ObjectWeb C−JDBC offre l'équilibrage de charge et la haute disponibilité au niveau de la base de données. L'utilisation de C−JDBC est transparente à l'application (JOnAS dans notre cas), puisqu'il est vu comme un driver JDBC standard. Cependant, c'est ce "driver" qui implémente les mécanismes propres à la mise en cluster (Equilibrage de charge sur les lectures, "broadcast" sur les écritures). La base de données est distribuée et répliquée sur plusieurs noeuds, et C−JDBC répartit la charge des requêtes entre les différents noeuds. Une évalution de C−JDBC

par le benchmark TPC−W sur un cluster à 6 noeuds a montré un accroissement linéaire de la performance jusqu'à 6 noeuds.

En plus des solutions de mise en cluster, JOnAS fournit plusieurs mécanismes intrinsèques qui lui assurent une haute scalabilité et des performances accrues. Ces mécanismes incluent :

- Un pool d'instances de bean session sans état
- Un pool d'intances de bean entité, configurable pour chaque bean entité par son descripteur de déploiement
- Activation/passivation des beans entité, la passivation pouvant être contrôlée depuis la console d'administration
- Des pools de connexions pour des connecteurs JDBC, JMS, J2EE CA
- Un pool de threads pour les beans "Message−Driven"
- Un timeout peut être configuré lors du déploiement pour les beans session
- Le flag "shared" dans le descripteur de déploiement spécifique de chaque bean entité indique si la représentation persistante de ce bean entité est partagée par plusieurs serveurs ou applications, ou si elle est réservée au serveur JOnAS dans lequel il est chargé. Dans ce dernier cas, l'optimisation apportée par JOnAS consiste à ne pas recharger les données correspondantes entre les transactions.
- Le mécanisme habituel EJB 1.1 "isModified" (ou "Dirty") est disponible, pour éviter le stockage de données non modifiées.
- Une optimisation appelée "prefetching" permet la ré−utilisation par le conteneur EJB d'ensembles de résultats (ResultSet) obtenus lors de précédentes requêtes JDBC (cette optimisation réduit le nombre de requêtes SQL exécutées par le conteneur d'EJB).

Certains benchmarks et cas d'utilisations de JOnAS ont déjà prouvé la haute scalabilité de JOnAS (se référer aux résultats de Rubis ou au Cas d'Utilisation OpenUSS). Rubis est un benchmark pour les applications J2EE de commerce électronique, qui appartient maintenant au projet ObjectWeb JMOB (Java Middleware Open Benchmarking). OpenUSS est un portail universitaire opérationnel avec approximativement 10 000 utilisateurs.

Perspectives

En tant qu'implémentation opensource d'un serveur J2EE, JOnAS est en continuelle évolution pour satisfaire les besoins des utilisateurs et pour suivre l'évolution des standards. Les principales évolutions planifiées à l'heure actuelle sont les suivantes :

- L'objectif primordial de JOnAS est la conformité aux spécifications J2EE 1.4 .
- Début 2004, une nouvelle version du connecteur JDBC, conforme J2EE CA 1.5 (J2EE 1.4), sera disponible.
- Le support de la spécification EJB 2.1 (J2EE 1.4) sera disponible début 2004.
- Le support des "Web Services" sera complété par des outils de développement et une implémentation open source UDDI sera intégrée.
- Les APIs de déploiement conformes au spécifications JSR88 (J2EE 1.4) seront supportées début 2004, résultat du projet ishmael.
- L'administration de JOnAS sera enrichie du concept d'administration de domaines.
- Le projet ObjectWeb JOTM sera intégré dans le service Transactions de JOnAS. Ce sera la base pour le support futur de transactions imbriquées et de transactions "Web Services".

Démarrer avec JOnAS

L'objectif de ce manuel est de guider l'utilisateur tout au long de l'installation de JOnAS et de lui permettre de faire fonctionner une première application à base d'EJB. Il permet également d'aider à exécuter une application plus complexe utilisant les EJB pour accéder à une base de données.

Des informations complémentaires sur la configuration de JOnAS sont disponibles ici.

Installation de JOnAS

Où trouver JOnAS ?

La dernière version binaire stable de JOnAS est accessible sur le site suivant :

- Objectweb

La version binaire et les sources sont disponibles sur ce site. La version actuelle (ou toute version précédente) des sources JOnAS sont accessibles via CVS.

Télécharger JOnAS

La page de téléchargement de JOnAS vous permettra de choisir soit la dernière version binaire de JOnAS (JOnAS seul ou packagé avec Tomcat ou Jetty) soit les derniers codes source.

Toutes ces configurations sont disponibles soit dans des fichiers au format *.tgz* soit via des fichiers auto−installables *.exe* pour Windows.

Paramétrer l'environnement JOnAS

Sélectionner l'endroit pour installer JOnAS, par exemple your_install_dir.

Attention, si une précédente version de JOnAS est déjà installée dans ce répertoire, la nouvelle installation s'enregistrera par dessus les fichiers existants, et votre précédente configuration sera perdue. Il est donc recommandé de sauvegarder ces fichiers de configuration avant de commencer une installation. Il est également recommandé de paramétrer les fichiers de configuration dans un répertoire différent grâce à la variable d'environnement JONAS_BASE, tel qu'expliqué dans le Guide de Configuration.

L'installation de JOnAS consiste à décrompresser les fichiers téléchargés. Pour commencer le processus d'installation, aller dans le répertoire dans lequel JOnAS doit être installé (e.g., `your_install_dir`) et décompresser les fichiers téléchargés. Pour cela, utiliser `gunzip` et `tar −xvf` si vous êtes sous unix, ou l'utilitaire *winzip* si vous êtes sous Windows.

Une fois JOnAS installé, les deux variables suivantes doivent être configurées en fonction de votre environnment :

- JONAS_ROOT = `your_install_dir`/JONAS où `your_install_dir` est le répertoire dans lequel JOnAS a été installé,
- PATH = $JONAS_ROOT/bin/unix sous Unix ou %JONAS_ROOT%\bin\nt sous Windows.

Si vous utilisez la version JOnAS seul (sans Tomcat ou Jetty), vous devez exécuter '`ant install`' dans le répertoire JOnAS_ROOT afin de reconstruire les bibliothèques globales propres à votre environnement. Cela doit aussi être fait à chaque fois que votre environnement web change (e.g. passer de CATALINA vers JETTY). Cette étape n'est pas nécessaire si vous utilisez les packages JOnAS−Tomcat et JOnAS−Jetty.

En outre, si RMI/IIOP doit être utilisé, les bibliothèques globales doivent être reconstruites via la commande '`ant installiiop`' dans le répertoire JOnAS_ROOT. Ceci est valable pour tous les packages JOnAS. Se référer également au chapitre "Choix du protocole de communication" du Guide de Configuration.

JOnAS est maintenant prêt pour éxécuter sa première application à base d'EJB.

Exécuter une première application à base d'EJB

Exemples fournis avec JOnAS

La distribution JOnAS fournit plusieurs exemples disponibles dans le répertoire `$JONAS_ROOT/examples/src`. L'exemple situé dans le répertoire `sb` doit être exécuté en premier.

Cet exemple basique permettra de vérifier que JOnAS est correctement installé.

Dans cet exemple, un client java accède à un **Stateful Session Bean** et appelle la méthode *buy* du bean plusieurs fois à l'intérieur du scope de transactions.

Construire l'exemple sb

L'utilitaire de compilation Ant 1.5 build doit être utilisé pour construire les exemples. Remarque : quelques actions de Ant nécessitent un `bcel.jar` additionnel qui est disponible sur le site BCEL (pour aller directement à la page de téléchargement, cliquer ici). Si l'utilitaire *Ant 1.5* est installé sur votre système, vous pouvez construire les exemples JOnAS avec le fichier *build.xml* qui se trouve sous les répertoires *$JONAS_ROOT/examples* ou *$JONAS_ROOT/examples/src*.

Exécuter l'exemple sb

Deux processus sont mis en oeuvre dans cet exemple fourni avec la distribution JOnAS :

- Le Serveur JOnAS, au sein duquel les beans seront chargés, et
- le client java qui crée des instances des beans et appelle les méthodes métier de ceux−ci.

Pour exécuter l'exemple :

- Démarrer le serveur JOnAS :
  ```
  jonas start
  ```
  le message suivant s'affiche :
  ```
  The JOnAS Server 'jonas' version−number is ready
  ```
- Charger le jar contenant l'exemple sb afin de rendre les beans accessibles aux clients :
  ```
  jonas admin −a sb.jar
  ```
  le message suivant s'affiche :
  ```
  message−header : Op available
  ```
- Démarrer le client java dans un nouveau *xterm* Unix :
  ```
  jclient sb.ClientOp
  ```
- Ou démarrer le client java dans une console Windows :
  ```
  jclient sb.ClientOp
  ```
- Les informations suivantes s'affichent alors :
  ```
  Create a bean
  Start a first transaction
  First request on the new bean
  Second request on the bean
  Commit the transaction
  Start a second transaction
  Rollback the transaction
  Request outside any transaction
  ClientOp OK. Exiting.
  ```

  L'affichage de ces informations à l'écran signifie que la première application à base d'EJB a démarré correctement.

- Avant de quitter, assurez−vous de bien avoir arrêté le serveur JOnAS :
  ```
  jonas stop
  ```
- Ces instructions sont également accessibles dans le fichier **README** situé dans le répertoire de travail.

Comprendre pourquoi cela fonctionne

- Pour commencer, le `CLASSPATH` n'a pas besoin d'être paramétré parce que, lorsque le serveur JOnAS est démarré, le script `jonas` appelle la classe JOnAS `bootstrap`, qui charge toutes les classes nécessaires à JOnAS. Il est même recommandé d'avoir le moins d'éléments possible dans le CLASSPATH afin d'éviter tout conflit potentiel.
  Ensuite, le script client `jclient` est utilisé pour démarrer le client. Remarquez que les classes bean sont situées dans le répertoire `$JONAS_ROOT/examples/classes`. Si ce n'est pas le cas, il sera nécessaire d'appeler le script `jclient` avec l'option `−cp "$JONAS_ROOT/ejbjars/sb.jar"`.
- Le client a réussi à contacter le serveur JOnAS car il possède une référence distribuée préalablement enregistrée dans le service de nommage. Pour cela, le serveur et le client utilisent **JNDI**. Le fichier `carol.properties` contenant la configuration JNDI se situe dans le répertoire `$JONAS_ROOT/conf`. Le fichier `carol.properties` contient les paramètres jndi par défaut. Grâce à ces valeurs par défaut, le *registry* démarre sur le `localhost` avec le port par défaut (1099 pour RMI/JRMP).
  Par défaut, le *registry* est démarré sur la même JVM que le serveur JOnAS.

REMARQUE

jclient est le script qui est utilisé pour faciliter l'exécution d'un pur client java dans un environnement de développement.
Ce n'est pas la méthode recommandée par J2EE pour exécuter un client java. J2EE recommande d'utiliser un conteneur client J2EE. Des exemples de conteneur client sont disponibles dans les applications exemples `earsample` et `jaasclient` fournis avec la distribution JOnAS.

Des exemples plus complexes

Autres exemples JOnAS

Les applications exemples disponibles sous `$JONAS_ROOT/example/src` sont brièvement décrites ci−après :

1. **L'application exemple *eb*** utilise des **Entity beans**.
   Les deux beans partagent la même interface (`Account`): l'un utilise la persistance explicite codée dans le bean (bean−managed persistence), l'autre utilise la persistance implicite proposée par le container (container−managed persistence).
   C'est un bon exemple pour comprendre ce qui doit être fait et ce qui ne le doit pas, en matière de gestion de la persistance, suivant le mode de persistance choisi. Les implémentations CMP 1.1 et CMP 2.0 sont toutes deux fournies dans cette exemple.
2. L'application exemple **lb** utilise les **Entity bean avec des interfaces locales**.
   Un bean session, `Manager`, gère localement un entity bean, `Manac`, représentant un `Account`. C'est un bon exemple pour comprendre ce qui doit être fait pour un client local en présence d'un entity bean fournissant des interfaces locales.

3. Le répertoire **jms** contient un **Stateful Session bean** possédant des méthodes mettant en oeuvre des opérations JMS ainsi qu'un client récepteur de message pure JMS.
   Une descritpion complète de cet exemple est disponible ici dans le guide de l'utilisateur JMS.
4. Le répertoire **mailsb** contient un **SessionMailer** et un **MimePartDSMailer Stateful Session bean** avec des méthodes fournissant une façon de construire et d'envoyer des messages (mail).
5. **mdb/samplemdb** contient un **Message Driven bean** capable d'écouter un "topic" particulier et un MdbClient, qui est un client JMS pure, qui envoie 10 messages sur le "topic" en question.
   C'est un bon exemple pour comprendre comment écrire et utiliser les message−driven beans.
6. **mdb/sampleappli** contient deux **Message−Driven beans**, l'un écoutant un "topic" particulier (`StockHandlerBean`), l'autre écoutant une fil d'attente ou "Queue" (`OrderBean`); un **Entity bean** dont la persistance est gérée par le conteneur (`StockBean`); et un **Stateless Session bean** pour créer la table utilisée dans la base de données.
   L'application SampleAppliClient envoie plusieurs messages sur le "topic" concerné. Lors de la réception d'un message, le **StockHandlerBean** met à jour la base de données via le **StockBean** et envoie un message à la fil d'attente ("Queue"), le tout à l'intérieur d'un transaction globale. Tous les EJBs sont impliqués dans des transactions qui peuvent être validées (commit) ou annulées (rollback).
7. **L'application alarm** surveille des messages d'alarme générés de manière asynchrone via JMS. Elle implémente différentes techniques utilisées dans JOnAS :

   - un Entity Bean pour AlarmRecord, persistant dans la base de données,
   - un Session Bean pour permettre aux clients de visualiser les alarmes reçues,
   - un Message Driven Bean pour capter les alarmes,
   - JMS pour utiliser un "Topic" sur lequel les alarmes sont envoyées et reçues,
   - Tomcat ou Jetty (pour les pages JSP et les servlets),
   - la sécurité.

8. **l'application earsample** contient une application J2EE complète. Cet exemple contient un simple **stateful Session bean**, avec une synchronisation et de la sécurité.
   Ce bean est accédé à partir d'un composant servlet dans lequel l'utilisateur est authentifié, JOnAS contrôle l'accès aux méthodes du bean. La servlet exécute une multitude de consultations (resource−ref, resource−env−ref, env−entry, ejb−ref et ejb−local−ref) dans l'environnement java:comp/env pour illustrer comment le nommage uniforme J2EE fonctionne dans les servlets.
9. **cmp2** contient un exemple illustrant la plupart des concepts de CMP 2.0.
10. **jaasclient** contient un exemple de module de login JAAS illustrant les différentes méthodes d'authentification. Ceci s'applique à un client Java lourd.
    Il y a différents gestionnaires de callback (CallbackHandler) qui démontrent comment saisir l'identification via une ligne de commande, avec une boîte de dialogue, ou sans message de sollicitation (cas où le client Java contient son propre login/password dans le code).

Les répertoires correspondant contiennent des fichiers **README** expliquant comment construire et exécuter chaque exemple.

Un exemple avec accès à une base de données

L'exemple **eb** contient deux Entity beans gérant des objets `Account`.

Les deux beans partagent la même interface (Account); l'un emploie la persistance explicite, gérée par le Bean (BMP), l'autre la persistance implicite, gérée par le container (CMP). L'implémentation CMP par défaut est la version CMP 1.1. Une implémentation CMP 2.0 est également fournie et son utilisation est décrite dans le fichier README.

Avant d'exécuter cet exemple :

- configurer l'accès à la base de données,
- configurer votre environnement classpath,
- créer la table dans la base de données à utiliser pour cet exemple (BMP et CMP 1.1 uniquement).

Configurer l'accès à la base de données

Afin de pouvoir accéder à une base de données relationnelle, JOnAS créera et utilisera un object DataSource qui devra être configuré en fonction de la base de données utilisée.

Ces objets DataSource sont configurés via des fichiers de propriétés Java (properties). `$JONAS_ROOT/conf` contient des modèles pour configurer les objets DataSource pour les bases de données Oracle, InstantDB, InterBase, McKoi, MySQL ou PostgreSQL :

- `Oracle1.properties`,
- `InstantDB1.properties`,
- `InterBase1.properties`,
- `McKoi1.properties`,
- `MySQL.properties`,
- `PostgreSQL1.properties`

En fonction de votre base de données, vous pouvez paramétrer un de ces fichiers avec les valeurs correspondantes à votre installation. Ensuite la valeur de la propriété `jonas.service.dbm.datasources` du fichier `jonas.properties` doivent être mis à jour.

Par exemple:

```
jonas.service.dbm.datasources    Oracle1
```

pour le fichier `Oracle1.properties`.

La section "Configurer le service base de données" dans le Guide de Configuration fournit plus d'informations sur les objets DataSource et leur configuration.

Créer la table dans la base de données

`$JONAS_ROOT/examples/src/eb` contient un script SQL pour Oracle, `Account.sql`, et un autre pour

InstantDB, `Account.idb`. Si votre serveur Oracle est en marche, ou si InstantDB est correctement installé et configuré, vous pouvez créer une table pour l'application exemple (Ne rien créer si vous utilisez la CMP 2.0).

Exemple pour Oracle:

```
sqlplus user/passwd
SQL> @Account.sql
SQL> quit
```

Configurer le classpath

Les classes du driver JDBC doivent être accessibles à partir du classpath. Pour cela, mettre à jour le fichier `config_env` ou `config_env.bat`. Dans ce fichier, attribuer aux variables suivantes : `IDB_CLASSES`, `ORACLE_CLASSES`, `POSTGRE_CLASSES` ou `INTERBASE_CLASSES` les valeurs correspondantes à l'installation de votre base de données.

Vous avez aussi la possibilité de mettre directement les classes du driver JDBC dans votre CLASSPATH, ou de les déposer dans le répertoire `JONAS_ROOT/lib/ext`.

Construire l'exemple eb

L'utilitaire *Ant 1.5* est utilisé pour construire les exemples JOnAS en utilisant les fichiers *build.xml* situés dans les répertoires *$JONAS_ROOT/examples* ou *$JONAS_ROOT/examples/src*.
Pour cela, utiliser le shell script `build.sh` sous Unix, ou le script `build.bat` sous Windows.

Exécuter l'exemple eb

Ici également, deux processus sont impliqués :

- Le Serveur JOnAS, au sein duquel les beans seront chargés
- le client java qui crée des instances de beans et appelle les méthodes métier de ceux−ci.

Pour exécuter cet exemple :

- Démarrer le serveur JOnAS pour rendre les beans accessibles aux clients :

  ```
  jonas start
  jonas admin −a eb.jar
  ```

  Les informations suivantes s'affichent alors :

  ```
  The JOnAS Server 'jonas' version−number is ready and running on rmi

  message−header : AccountExpl available
  message−header : AccountImpl available
  ```

- Ensuite, démarrer les clients java sous d'autres *xterm* Unix :

  ```
  jclient eb.ClientAccount AccountImplHome
  ```

```
jclient eb.ClientAccount AccountExplHome
```

- Ou démarrer les clients java dans la console Windows :

```
jclient eb.ClientAccount AccountImplHome
jclient eb.ClientAccount AccountExplHome
```

- Si les informations suivantes s'affichent :

```
Getting a UserTransaction object from JNDI
Connecting to the AccountHome
Getting the list of existing accounts in database
101 Antoine de St Exupery 200.0
102 alexandre dumas fils 400.0
103 conan doyle 500.0
104 alfred de musset 100.0
105 phileas lebegue 350.0
106 alphonse de lamartine 650.0
Creating a new Account in database
Finding an Account by its number in database
Starting a first transaction, that will be committed
Starting a second transaction, that will be rolled back
Getting the new list of accounts in database
101 Antoine de St Exupery 200.0
102 alexandre dumas fils 300.0
103 conan doyle 500.0
104 alfred de musset 100.0
105 phileas lebegue 350.0
106 alphonse de lamartine 650.0
109 John Smith 100.0
Removing Account previously created in database
ClientAccount terminated
```

cela signifie que l'exemple *eb*. a été exécuté correctement.

- Avant de quitter, assurez−vous d'avoir bien arrété le serveur JOnAS :

```
jonas stop
```

Guide de Configuration

Le contenu de ce guide est le suivant :

Les règles de configuration de JOnAS

Ainsi qu'il est décrit dans le chapitre Démarrer avec JOnAS, JOnAS est pré−configuré avec *RMI/JRMP* pour les accès distants. Il est prêt à être utilisé tant qu'il n'est pas nécessaire d'avoir la visibilité sur d'autres classes que celles contenues dans la distribution JOnAS, localisée sous `$JONAS_ROOT/lib`.
Si l'on souhaite utiliser *JEREMIE* ou *RMI/IIOP* pour les accès distants, ou pour intégrer des classes java additionnelles (par exemple, des classes driver JDBC), des opérations de configuration complémentaires sont nécessaires, telles que le positionnement du numéro de port spécifique au registre.

La distribution JOnAS contient un certain nombre de fichiers de configuration sous le répertoire `$JONAS_ROOT/conf`. Ces fichiers peuvent être édités pour changer la configuration par défaut. Cependant, il est recommandé de placer dans un répertoire spécifique les fichiers de configuration propres à une application particulière tournant sur JOnAS. Ceci se fait par le moyen d'une variable d'environnement supplémentaire appelée ***JONAS_BASE***.

JONAS_BASE

La variable d'environnement `JONAS_BASE` a été introduite dans JOnAS pour la version 3.1. A partir de celle−ci, la règle de configuration des versions précédentes est remplacée par une nouvelle, très simple :

Les fichiers de configuration JOnAS sont lus depuis le répertoire`$JONAS_BASE/conf`.
Si la variable `JONAS_BASE` n'est pas définie, elle est automatiquement initialisée à `$JONAS_ROOT`.

Il y a deux façons d'utiliser la variable `JONAS_BASE`:

1. Effectuer les actions suivantes :
   - Créer un nouveau répertoire et initialiser `JONAS_BASE` avec le chemin d'accès à ce répertoire.
   - Créer les sous−répertoires suivants dans `$JONAS_BASE`:
     - conf
     - ejbjars
     - apps
     - webapps
     - rars
     - logs
   - Copier les fichiers de configuration localisés dans `$JONAS_ROOT/conf` vers `$JONAS_BASE/conf`. Ensuite, adapter ces fichiers de configuration aux besoins de votre application, en suivant les explications fournies dans la suite du document ;
2. Effectuer les actions suivantes:
   - Initialiser `$JONAS_BASE` avec un chemin d'accès donné,
   - Se positionner dans le répertoire `$JONAS_ROOT`, et exécuter la commande : `ant create_jonasbase`. Ceci va copier tous les fichiers requis et créer tous les répertoires nécessaires.

Les fichiers `build.xml` fournis avec les exemples JOnAS intègrent la variable `JONAS_BASE`. Si cette variable d'environnement est définie avant de construire et d'installer les exemples, les fichiers archives générés sont installés

dans le sous−répertoire approprié de $JONAS_BASE. Par exemple, les fichiers ejb−jar files correspondant aux exemples types de $JONAS_ROOT/examples/src/ sont localisés dans le répertoire $JONAS_BASE/ejbjars.

Configurer l'environnement JOnAS

Le fichier de configuration JOnAS

Le serveur JOnAS se configure à l'aide du fichier de configuration ***jonas.properties***. Il contient une liste de couples clé/valeur présentée dans le format des fichiers de propriétés Java.

Une configuration par défaut est fournie dans $JONAS_ROOT/conf/jonas.properties. Ce fichier recense toutes les propriétés possibles avec leurs valeurs par défaut. Ce fichier de configuration est obligatoire. Le serveur JOnAS le recherche au lancement dans le répertoire $JONAS_BASE/conf ($JONAS_ROOT/conf si la variable $JONAS_BASE n'est pas définie).

La plupart des propriétés concernent les services JOnAS qui peuvent être lancés avec le serveur JOnAS. Ces propriétés sont décrites en détail dans le chapitre Configurer les services JOnAS.
La propriété jonas.orb.port est indépendante de tout service. Elle identifie le numéro de port sur lequel les objets distants reçoivent les appels. Sa valeur par défaut est **0**, ce qui signifie qu'un port anonyme est choisi. Lorsque le serveur JOnAS est utilisé derrière un firewall, cette propriété peut être positionnée avec un numéro de port spécifique.

NB : à partir de la version JOnAS 3.1. , la proprété jonas.name n'est plus supportée. Le nom du serveur peut être spécifié sur la ligne de commande ***jonas*** en utilisant l'option −n (***−n name***). Si le nom n'est pas spécifié, la valeur par défaut est **jonas**.

Lorsque plusieurs serveurs JOnAS tournent de façon simultanée, il est intéressant de leur affecter à chacun un nom différent, dans le but de simplifier leur administration à l'aide des Outils d'administration de JOnAS.

Notez également qu'il est possible de définir des propriétés de configuration directement sur la ligne de commande : (java −D...).

La commande ***jonas check*** présente l'état courant de la configuration de JOnAS.

Les scripts de configuration

La distribution JOnAS contient deux scripts de configuration :

- $JONAS_ROOT/bin/unix/setenv et $JONAS_ROOT/bin/unix/config_env, sur Unix

  Ce script de configuration positionne les variables d'environnement utiles au démarrage de JAVA ($JAVA et $JAVAC). Il ajoute $JONAS_BASE/conf à la variable $CLASSPATH si $JONAS_BASE est positionné, ou $JONAS_ROOT/conf dans le cas contraire. Ce script est appelé par la plupart des autres scripts (jclient, jonas, newbean, registry, GenIC).

- `%JONAS_ROOT%\bin\nt\setenv.bat` et `%JONAS_ROOT%\bin\nt\config_env.bat`, sur Windows

  Ce script est utilisé pour ajouter des fichiers `.jar` nécessaires à la variable `$CLASSPATH`. Ce script est appelé par le script `jonas`.

Cependant, lorsqu'il est nécessaire d'avoir la visibilité sur certains fichiers `.jar`, la meilleure pratique est de mettre à jour le fichier ***`config_env`*** . Par exemple, pour voir certaines classes JDBC, une ou plusieurs des variables `IDB_CLASSES`, `ORACLE_CLASSES`, `POSTGRE_CLASSES`, `INTERBASE_CLASSES` de ce fichier doivent être mises à jour.

Une autre façon d'ajouter un `.jar` dans le chemin d'accès aux classes de votre serveur JOnAS server est de l'insérer à la fin du fichier ***`config_env`*** :

```
CLASSPATH=<The_Path_To_Your_Jar>$SPS$CLASSPATH
```

Notez qu'une variable d'environnement supplémentaire, appelée `XTRA_CLASSPATH` peut être définie pour charger des classes spécifiques lors du démarrage du serveur JOnAS. Se référer au chapitre Chargeur de classe initialiseur.

Configurer le Protocole de Communication et l'interface JNDI

Choisir le Protocole

Habituellement, l'accès à l'interface JNDI est lié au fichier ***`jndi.properties`*** qui doit être accessible depuis le chemin d'accès aux classes (classpath). Le fonctionnement est légèrement différent avec JOnAS. A partir de la version 3.1.2 de JOnAS, le mode multi−protocoles est supporté grace à l'intégration du composant CAROL. Actuellement, RMI/JRMP, RMI/IIOP, JEREMIE, et CMI (protocole en mode cluster) sont supportés, en changeant la configuration. De nouveaux protocoles pourraient être supportés dans le futur. Dans les versions antérieures à JOnAS 3.1.2, il était nécessaire de reconstruire JOnAS pour passer de l'un à l'autre (p. ex., de RMI/JRMP à JEREMIE), et de changer la valeur de la variable d'environnement OBJECTWEB_ORB; OBJECTWEB_ORB n'est désormais plus utilisée. Cette configuration est maintenant prise en compte dans le fichier ***`carol.properties`*** (qui inclue les anciennes particularités du fichier `jndi.properties`). Ce fichier est fourni avec la distribution JOnAS, dans le répertoire `$JONAS_ROOT/conf`.

Les protocoles de communication supportés sont les suivants :

- ***RMI/JRMP*** est l'implémentation du JRE de RMI sur le protocole JRMP. C'est le protocole de communication par défaut.
- ***JEREMIE*** est une implémentation de RMI proposée par le projet Objectweb Jonathan. Il fournit une optimisation des appels locaux (appels RMI dans la même JVM). *JEREMIE* possède son propre fichier de configuration ***`jonathan.xml`*** delivré dans le répertoire ***`$JONAS_ROOT/conf`***. Généralement, aucune modification n'est nécessaire, il suffit de s'assurer que le fichier est présent sous ***`$JONAS_BASE/conf`*** si la variable `$JONAS_BASE` est positionnée.
- ***RMI/IIOP*** est l'implémentation de RMI sur le protocole IIOP.

- ***CMI*** (Cluster Method Invocation) est le protocole de communication utilisé par JOnAS dans le cas de configuration en mode cluster. Notez que ce protocole est basé sur JRMP.

Le contenu du fichier ***carol.properties*** est le suivant:

```
# jonas rmi activation (jrmp, iiop, jeremie, or cmi)
carol.protocols=jrmp
#carol.protocols=cmi
#carol.protocols=iiop
#carol.protocols=jeremie
# RMI JRMP URL
carol.jrmp.url=rmi://localhost:1099

# RMI JEREMIE URL
carol.jeremie.url=jrmi://localhost:12340

# RMI IIOP URL
carol.iiop.url=iiop://localhost:2000

# CMI URL
carol.cmi.url=cmi://localhost:2001


# general rules for jndi
carol.jndi.java.naming.factory.url.pkgs=org.objectweb.jonas.naming
```

CAROL peut être personnalisé en éditant le fichier `$JONAS_BASE/conf/carol.properties` pour 1) choisir le protocole parmi les propriétés `carol.protocols`, 2) changer la valeur de `localhost` pour celle du serveur sur lequel le registre va être exécuté, 3) changer le numéro de port standard. Si le numéro de port standard est changé, le registre doit être lancé avec ce numéro de port en paramètre, `registry <Your_Portnumber>`, lorsque le registre n'est pas chargé dans le serveur JOnAS.

Du fait d'une incompatibilité entre Jeremie et rmi/iiop, il existe actuellement une ***restriction*** ; Il n'existe pas d'option de configuration permettant de choisir entre ces deux protocoles lorsque l'on construit un serveur JOnAS. Par défaut, JOnAS permet uniquement de choisir entre `rmi/jrmp`, `Jeremie,` ou `cmi`. Pour utiliser `rmi/iiop`, il est nécessaire de lancer la ligne de commande "`ant installiiop`" depuis le répertoire $JONAS_ROOT. Ceci va permettre un choix entre `rmi/jrmp`, `rmi/iiop` ou `cmi`.

Propagation des contextes de sécurité et de transaction

Pour implémenter la sécurité et le mode transactionnel des EJB, JOnAS repose sur la couche communication pour propager les contextes transactionnel et de sécurité au travers des appels méthodes. Par défaut, le protocole de communication est configuré pour la propagation de contexte. Cependant, cette configuration peut être changée en la désactivant pour la sécurité et/ou les transactions ; Ceci se fait principalement lorsque l'on veut gagner en performance. La propagation de contexte peut être configurée dans le fichier ***jonas.properties*** en positionant les propriétés `jonas.security.propagation` et `jonas.transaction.propagation` à `true` ou `false`:

```
# valider la propagation du contexte de sécurité
jonas.security.propagation              true

# valider la propagation du contexte transactionnel
jonas.transaction.propagation           true
```

Notez que l'attribut `secpropag` de la tâche JOnAS ejbjar ANT ainsi que l'option `-secpropag` de GenIC ne sont plus utilisées. Elles étaient auparavant utilisées dans les versions de JOnAS antérieures à la version 3.1.2 pour demander la propagation du contexte de sécurité.

Déploiement Multi-protocole (GenIC)

Il est nécessaire de spécifier à l'outil de déploiement JOnAS (GenIC) quels stubs de protocoles (pour les invocations à distance) doivent être générés. Le fait de choisir plusieurs protocoles va éliminer le besoin de redéployer les EJBs lorsque l'on voudra passer de l'un à l'autre. Par défaut, GenIC génère des stubs pour `rmi/jrmp` et `Jeremie`. Pour changer cette configuration, il suffit de lancer GenIC avec l'option `-protocols` suivi d'une liste de protocoles séparés par une virgule (choisi parmi jeremie, jrmp, iiop, cmi), p. ex. :

```
GenIC -protocols jrmp,jeremie,iiop
```

Cette liste de protocoles peut également être spécifiée pour la tâche JOnAS ejbjar ANT :

```
<jonas destdir="${dist.ejbjars.dir}"
    jonasroot="${jonas.root}"
    protocols="jrmp,jeremie,iiop"
    keepgenerated="true"
    verbose="${verbose}"
    mappernames="${mapper.names}"
    additionalargs="${genicargs}">
 </jonas>
```

Configurer le système de trace (monolog)

Depuis la version 2.5 de JOnAS, le système de trace est basé sur Monolog, l'API standard pour les projets Objectweb. Configurer les messages de trace dans JOnAS peut se faire de deux façons différentes :

- Changer le fichier trace.properties pour configurer la trace de façon statique, avant le lancement du serveur JOnAS,
- Utiliser la commande jonas admin ou l'outil d'administration JonasAdmin pour configurer les traces de façon dynamique, pendant que le serveur JOnAS est en cours de fonctionnement

Notez que les requêtes SQL envoyées à une base de données peuvent être facilement tracées en utilisant le système de trace de JOnAS et l'outil P6Spy intégré. Les étapes de configuration sont décrites dans la section "Configurer les sources de données JDBC".

Syntaxe de trace.properties

Un fichier standard est fourni dans `$JONAS_ROOT/conf/trace.properties`. Utilisez la variable CLASSPATH pour accéder à ce fichier.

La documentation monolog fournit plus de détails sur la façon de configurer les traces. Monolog est basée sur une API standard de trace (actuellement, log4j). Il est possible de définir des Loggers, chacun étant confié à un gestionnaire (handler).
Un handler représente une sortie, est identifié par un nom, possède un type et quelques propriétés additionnelles. Deux handlers ont été définis dans le fichier trace.properties :

- **tty** représente la sortie standard sur une console, sans en−têtes.
- **logf** est un handler pour l'impression de messages dans un fichier.

Chaque handler peut définir l'en−tête qu'il va utiliser, le type de trace (console, fichier, fichier cyclique), ainsi que le nom de fichier.

Notez que si le tag "automatic" est spécifié comme nom de fichier de sortie, JOnAS va remplacer ce tag par un fichier pointant vers `$JONAS_BASE/logs/<jonas_name_server>−<timestamp>.log`.
Le handler logf, qui est inclu dans JOnAS, utilise ce tag "automatique".

Les Loggers sont identifiés par des noms structurés de façon arborescente. La racine de l'arbre est appelée **root**.

A chaque Logger est associé un "Topic". Les noms de Topic sont habituellement basés sur le nom du package.
Chaque logger peut définir le handler qu'il va utiliser, et le niveau de trace à l'aide des quatre valeurs suivantes :

- **ERROR** : erreurs. Devraient toujours être imprimées.
- **WARN** : avertissement. Devraient être imprimés.
- **INFO** : messages informatifs, peu utilisés dans JOnAS (par exemple: résultats de tests).
- **DEBUG** : messages de mise au point. Devrait être utilisé uniquement pour la mise au point.

Si rien n'a été défini concernant un logger, il va utiliser les propriétés définies pour ses parents.

Le code de JOnAS est écrit en utilisant l'API monolog et peut utiliser le handler **tty**.

Exemple de positionnement du niveau DEBUG pour le logger utilisé dans le module jonas_ejb:

```
logger.org.objectweb.jonas_ejb.level DEBUG
```

Exemple de positionnement de la sortie des traces JOnAS simultanément à la console et dans le fichier `/tmp/jonas/log`:

```
handler.logf.output /tmp/jonas.log
logger.org.objectweb.jonas.handler.0 tty
logger.org.objectweb.jonas.handler.1 logf
```

Exemple de positionnement de la sortie des traces JOnAS dans un fichier du répertoire `$JONAS_BASE/logs/` :

```
handler.logf.output automatic
logger.org.objectweb.jonas.handler.0 logf
```

Configurer les Services JOnAS

JOnAS peut être vu comme un ensemble de services intégrés et administrables lancés au démarrage du serveur. JOnAS est également capable de lancer des services externes, qui peuvent être définis ainsi qu'il est détaillé dans le chapitre Création d'un service JOnAS.

Voici une liste de services intégrés à JOnAS :

- **registry**: ce service est utilisé pour relier les objets distants et les ressources qui seront ultérieurement accédées par JNDI. Il est automatiquement chargé avant tout autre service au lancement de JOnAS.
- **ejb**: Le *Service Conteneur EJB* est le service supportant les composants applicatifs EJB.
- **web**: Le *Service Conteneur WEB* est le service supportant les composants web (tels les Servlets et JSP). A l'heure actuelle, JOnAS fournit une implémentation de ce service pour Tomcat et Jetty.
- ws: le service WebServices est le service supportant les WebServices (publication WSDL).
- **ear**: le *service EAR* est le service supportant les applications J2EE.
- **jmx**: ce service est nécessaire pour administrer le serveur et les services JOnAS à l'aide de la console d'administration basée sur JMX. Actuellement, JOnAS peut être configuré pour utiliser une des deux implémentations JMX suivantes :
  - L'Implémentation Référence de Sun
  - MX4J
- **security**: Ce service est nécessaire pour mettre en oeuvre la sécurité lors du fonctionnement de JOnAS.
- **jtm**: le *Service d'administration des transactions* est utilisé pour supporter le mode transactionnel distribué. C'est le seul service obligatoire de JOnAS.
- **dbm**: le *service base de données* est nécessaire aux composants applicatifs accédant une ou plusieurs bases de données.
- **resource**: ce service est nécessaire pour accéder aux *Connecteurs* conformes à la *Spécification d'architecture de connecteurs J2EE*.
- **jms**: ce service est nécessaire aux composants applicatifs utilisant l'API standard *Java Message Service*. Il est également obligatoire lorsque l'on utilise des beans message−driven.
- **mail**: le *Service Mail* est nécessaire pour les applications devant envoyer des e−mails.

Ces services peuvent être administrés par la console d'administration `JonasAdmin`, application basée sur des Servlets utilisant la technologie JMX. Notez que JonasAdmin peut uniquement administrer les services disponibles à un moment donné. Actuellement, il n'est pas possible de lancer un service après le démarrage du serveur.

Les services disponibles dans un serveur JOnAS sont ceux spécifiés dans le fichier de configuration JOnAS. La propriété `jonas.services` dans le fichier `jonas.properties` doit contenir une liste des noms de services

requis. Actuellement, ces services sont lancés **dans l'ordre** dans lequel ils apparaissent dans la liste. Les contraintes suivantes doivent donc être respectées :

- **jmx** doit précéder tous les autres services de la liste (excepté **registry**) pour permettre l'administration de ces services.
- **jtm** doit précéder les services suivants : **dbm**, **resource**, et **jms**.
- **security** doit être après **dbm**, puisqu'il utilise des sources de données.
- Les services utilisés par les composants applicatifs doivent être listés avant le service conteneur utilisé pour déployer ces composants. Par exemple, si l'application contient des EJBs qui nécessitent un accès base de données, **dbm** doit précéder **ejb** dans la liste des services requis.

Exemple:

```
jonas.services    registry,jmx,jtm,dbm,security,resource,jms,mail,ejb,ws,web,ear
```

Le service **registry** peut être omis de la liste puisqu'il est chargé de façon automatique s'il n'a pas déjà été lancé par un précédent serveur d'application. Depuis la version 3.1 de JOnAS, ceci est également vrai pour **jmx**, ce service étant automatiquement lancé après le service **registry**. Les services **dbm**, **resource**, et **jms** sont listés après **jtm**.
Les composants applicatifs déployés sur ce serveur peuvent utiliser la Messagerie Java et le Java Mail puisque **jms** et **mail** sont listés avant **ejb**.

Les paramètres de configuration pour les services se trouvent dans le fichier `jonas.properties`, et respectent strictement la convention de nommage suivante : un service **XX** sera configuré avec un ensemble de propriétés :

```
jonas.service.XX.foo   valeur1
    jonas.service.XX.bar   valeur2
```

Configurer le Service Registre

Ce service est utilisé pour accéder au registre RMI, au registre Jeremie, au registre CMI, ou au CosNaming (iiop), suivant la configuration des protocoles de communication définie dans ***`carol.properties`***. Pour plus de détail, se référer à la section "Configurer le Protocole de Communication et l'interface JNDI."

Il y a plusieurs modes de lancement du service Registre suivant la valeur de la propriété JOnAS `jonas.service.registry.mode`. Les différentes valeurs possibles sont les suivantes :

- `collocated`: le Registre est chargé dans la même JVM que le serveur JOnAS,
- `remote`: le Registre doit être chargé avant le serveur JOnAS dans une JVM distincte,
- `automatic`: le Registre est chargé dans la même JVM que le serveur JOnAS, s'il n'est pas déjà démarré. C'est la valeur par défaut.

Le numéro de port sur lequel le registre est chargé est défini dans le fichier `carol.properties`.

Configurer le Service Conteneur EJB

Ce service fourni les conteneurs d'EJB pour les composants EJB. En ce sens, c'est le service principal de JOnAS.

Un conteneur d'EJB peut être créé depuis un fichier `ejb-jar` file en utilisant une des fonctionnalités suivantes :

- Le nom du fichier `ejb-jar` file correspondant est listé dans la propriété `jonas.service.ejb.descriptors` du fichier `jonas.properties`. Si le nom du fichier ne contient pas un path absolu, il doit être présent dans le répertoire **`$JONAS_BASE/ejbjars/`**. Le conteneur est créé lorsque le serveur JOnAS est lancé.
  Par exemple :

  ```
  jonas.service.ejb.descriptors  Bank.jar
  ```

  Dans cet exemple, le *service Conteneur* créera un conteneur pour le fichier `ejb-jar` de nom `Bank.jar`. Ce fichier sera recherché dans le répertoire **`$JONAS_BASE/ejbjars/`**.
- Une autre façon de créer automatiquement un conteneur d'EJB au lancement du serveur est de placer les fichiers `ejb-jar` dans un répertoire *autoload*. Le nom de ce répertoire est spécifié un utilisant la propriété `jonas.service.ejb.autoloaddir` dans le fichier `jonas.properties`.
- Les conteneurs d'EJB peuvent être créés dynamiquement depuis les fichiers `ejb-jar` en utilisant l'outil `JonasAdmin`.

JOnAS permet également le chargement de composants EJB non déployés. Le nom du fichier xml contenant le descripteur de déploiement des EJB doit être spécifié dans la propriété `jonas.service.ejb.descriptors`. Notez que le serveur JOnAS doit alors avoir accès aux classes du composant, ce qui peut se faire en utilisant la variable d'environnement `XTRA_CLASSPATH` (voir le chapitre consacré au Chargeur de classe initialiseur).

Configurer le Conteneur WEB

Ce service fournit les conteneurs WEB pour les composants WEB utilisés par les applications J2EE.
JOnAS fournit deux implémentations de ce service : une pour Jetty 4.2.x, une pour Tomcat 4.1.x.
Il est nécessaire de lancer ce service, pour utiliser l'outil `JonasAdmin`.

Si aucune des variables d'environnement `$JETTY_HOME` ou `$CATALINA_HOME` n'est positionnée, le service conteneur Web ne peut être chargé au lancement de JOnAS, et un avertissement est affiché.

Un conteneur WEB est créé depuis un fichier `war`. Si le nom du fichier ne contient pas un chemin d'accès absolu, il doit être localisé dans le répertoire **`$JONAS_BASE/webapps/`** .

Pour faire en sorte que les conteneurs WEB soient créés au lancement du serveur, il faut renseigner la propriété `jonas.service.web.descriptors` dans le fichier `jonas.properties`.
Par exemple :

```
jonas.service.web.descriptors  Bank.war
```

Dans cet exemple, le *Service Conteneur WEB* va créer un conteneur depuis le fichier dénommé `Bank.war`. Il recherchera ce fichier sous le répertoire **`$JONAS_BASE/webapps/`**.

Utiliser directement des répertoires webapp au chargement plutôt que de packager un nouveau fichier war à chaque fois peut améliorer le processus de développement. Vous pouvez remplacer les anciennes classes par les nouvelles classes compilées, recharger les servlets dans le navigateur et le résultat peut immédiatement être vu. Ceci est vrai également pour les JSPs. Notez que ces fonctionnalités de chargement dynamique peuvent être désactivées dans la configuration du conteneur web (Jetty or Tomcat) pour la mise en production.
Exemple d'utilisation du répertoire **`jonasAdmin/webapp`** à la place du fichier **`jonasAdmin.war`**

- Se positionner dans le répertoire `JONAS_BASE/webapps/autoload`
- Créer un nouveau répertoire (par exemple, jonasAdmin) : `JONAS_BASE/webapps/autoload/jonasAdmin`
- Déplacer le fichier `jonasAdmin.war` dans ce répertoire
- Extraire le contenu du fichier war dans le répertoire courant, et détruire ensuite le fichier war
- Au prochain démarrage de JOnAS, le répertoire webapp sera utilisé à la place du fichier war. Modifiez la jsp et observez la prise en compte des modifications.

Les conteneurs WEB peuvent également être créés dynamiquement depuis les fichiers `war` en utilisant l'outil `JonasAdmin`.

Configurer le Service WebServices

A. Choisir un moteur de service Web

A l'heure actuelle, une seule implémentation des WebServices est disponible : L'implémentation d' Axis. Dans le futur, une implémentation Glue pourra facilement être réalisée.

jonas.properties :

```
#...

# the fully qualifier name of the service class
jonas.service.ws.class org.objectweb.jonas.ws.AxisWSServiceImpl

#...
```

B. Choisir un ou plusieurs Handler(s) WSDL

Les Handlers WSDL sont utilisés pour localiser et publier vos documents WSDL. Vous pouvez utiliser plusieurs Handlers WSDL, il suffit pour cela de les définir dans les jonas.properties.

Exemple : si vous souhaitez publier un WSDL dans le système de fichier local, utilisez le handler FileWSDLHandler

jonas.properties :

```
#...

# a list of comma separated WSDL Handlers
jonas.service.ws.wsdlhandlers file1
# Configuration of the file WSDL Handler
jonas.service.ws.file1.class org.objectweb.jonas.ws.handler.FileWSDLHandler
# Specify String params for WSDLHandler creation jonas.service.ws.file1.params
location
# Make sure that user that run JOnAS have rad/write access in this directory
jonas.service.ws.file1.location /path/to/directory/where/store/wsdls

#...
```

Configurer le Service EAR

Le *Service EAR* permet le déploiement d'applications J2EE complètes (incluant les fichiers `ejb-jar` et `war` archivés dans le fichier `ear`). Ce service est basé sur les services *Conteneur WEB* et *Conteneur EJB*. Le *Service Conteneur Web* est utilisé pour déployer les `wars` inclus dans l'application J2EE, tandis que le Service Conteneur EJB est utilisé pour déployer les `ejb-jars` inclus dans l'application J2EE

Ce service peut être configuré en positionnant la propriété `jonas.service.ear.descriptors` dans le fichier `jonas.properties`. Cette propriété précise la liste des `ears` à déployer au lancement de JOnAS.

L'outil `JonasAdmin` peut être utilisé pour déployer dynamiquement les composants applicatifs J2EE depuis un fichier `ear`.

Lorsque l'on spécifie un chemin d'accès relatif pour les noms de fichiers `ear`, le fichier doit être localisé dans le répertoire **`$JONAS_BASE/apps/`**.
Par exemple :

```
jonas.service.ear.descriptors  Bank.ear
```

Dans cet exemple, *le service EAR* va déployer le fichier `ear` `Bank.ear`. Il recherchera ce fichier sous le répertoire **`$JONAS_BASE/apps/`**.

Configurer le Service transaction

Le *Service Transaction* est utilisé par le *Service Conteneur* pour administrer les transactions des composants EJB, conformément aux directives contenues dans le descripteur de déploiement. Ce service est obligatoire. Le *Service Transaction* utilise un *Gestionaire de Transaction* qui peut être local ou chargé dans une autre JVM (un *Gestionnaire de Transaction* distant). Typiquement, lorsque plusieurs serveurs JOnAS fonctionnent ensemble, l'un des *Services Transaction* doit être considéré comme le *maître* et les autres comme *esclaves*. Les esclaves doivent être configurés comme s'ils fonctionnaient avec un *Gestionnaire de Transaction* distant.
Voici un exemple de configuration dans le cas où l'on utilise deux serveurs : l'un nommé TM pour lequel un *Service Transaction* autonome va s'exécuter, l'autre nommé EJB utilisé pour déployer un conteneur d'EJB.

```
jonas.name               TM
jonas.services           jtm
jonas.service.jtm.remote false
```

et

```
jonas.name                    EJB
jonas.services                jmx,security,jtm,dbm,ejb
jonas.service.jtm.remote      true
jonas.service.ejb.descriptors foo.jar
```

Une autre option de configuration possible est de définir la valeur du time−out sur les transactions, en secondes, par la propriété `jonas.service.jtm.timeout`.
La configuration par défaut est la suivante :

```
 jonas.service.jtm.timeout   60
```

Configurer le Service Base de Données

La description des nouveaux connecteurs JDBC, en remplacement du service Base de Données, se trouve dans la section Configurer les Connecteurs JDBC.

Pour permettre un accès à une ou plusieurs bases de données relationnelles (P. ex. . Oracle, InstantDB, PostgreSQL, ...), JOnAS va créer et utiliser des objets Datasource (source de données). De tels objets DataSource doivent être configurés en fonction du type de base de données utilisé pour gérer la persistance des beans. Les objets DataSource et leur configuration sont détaillés dans la section "Configurer les sources de données JDBC" .

Les sections suivantes expliquent brièvement comment configurer un objet DataSource pour votre base de donnée, de façon à faire fonctionner l'exemple de Bean Entité sur votre plate−forme.

Notez que les requêtes SQL envoyées à la base de donnée peuvent facilement être tracées en utilisant le système de

trace JOnAS et l'outil intégré P6Spy. Les différentes étapes de cette configuration sont décrites dans la section "Configurer les sources de données JDBC".

Configuration d'Oracle pour l'exemple proposé

Un modèle de fichier Oracle1.properties est fourni dans le répertoire d'installation. Ce fichier est utilisé pour définir un objet *DataSource*, appelé `Oracle1`. Ce modèle doit être mis à jour avec les valeurs correspondant à votre propre installation. Les champs sont les suivants:

datasource.name	Nom JNDI du DataSource: Le nom utilisé dans l'exemple est `jdbc_1`.
datasource.url	L'URL base de donnée JDBC : pour le driver Oracle JDBC "thin" (léger), `jdbc:oracle:thin:@`*`hostname`*`:`*`sql*net_port_number`*`:`*`ORACLE_SID`* En cas d'utilisation d'un driver JDBC Oracle OCI, l'URL est `jdbc:oracle:oci7:` ou `jdbc:oracle:oci8:`
datasource.classname	Nom de la classe implémentant le driver Oracle JDBC : `oracle.jdbc.driver.OracleDriver`
datasource.mapper	Adapteur (JORM), obligatoire uniquement pour la CMP2.0 (Plus de détail dans la section Configurer les sources de données JDBC): `rdb.oracle8` pour Oracle 8 et les versions antérieures
datasource.username	Nom de l'utilisateur de la base de données
datasource.password	Mot de passe de l'utilisateur de la base de données

Pour que la plate−forme EJB puisse créer l'objet Datasource correspondant, son nom Oracle1 (et non le nom JNDI) doit être dans le fichier `jonas.properties`, sur la ligne jonas.service.dbm.datasources:

```
jonas.service.dbm.datasources      Oracle1
```

Il peut y avoir plusieurs objets DataSource définis pour un même serveur EJB ; il y aura alors plusieurs fichiers *dataSourceName*.properties ainsi qu'une liste des noms Datasource sur la ligne jonas.service.dbm.datasources du fichier jonas.properties :

```
jonas.service.dbm.datasources      Oracle1, Oracle2, InstantDB1
```

Pour créer la table utilisée en exemple dans le script SQL `examples/src/eb/Account.sql`, le serveur Oracle doit tourner avec le driver JDBC installé (les drivers JDBC d'Oracle peuvent être téléchargés sur leur site Web).
Par exemple:

```
sqlplus user/passwd
SQL> @Account.sql
SQL> quit
```

Les classes du driver JDBC doivent être accessibles depuis le chemin d'accès classpath. Pour ceci, mettez à jour le fichier *config_env* ; (`$JONAS_ROOT/bin/unix/config_env` sur Unix, ou `%JONAS_ROOT%\bin\nt\config_env.bat` sur Windows).

Configurer InstantDB pour l'exemple fourni

Un modèle de fichier InstantDB1.properties est fourni dans le répertoire d'installation. Ce fichier est utilisé pour définir un objet Datasource nommé InstantDB1. Ce modèle doit être mis à jour avec les valeurs correspondant à votre installation. Les différents champs sont les suivants :

datasource.name	Nom JNDI du DataSource: Le nom utilisé dans l'exemple est `jdbc_1`.
datasource.url	L'URL base de donnée JDBC, pour InstantDB : `jdbc:idb=db1.prp.`
datasource.classname	Nom de la classe implémentant le driver JDBC: `org.enhydra.instantdb.jdbc.idbDriver`

Pour que la plate−forme EJB crée l'objet Datasource correspondant, son nom InstantDB (et non le nom JNDI) doit être dans le fichier ***jonas.properties***, sur la ligne jonas.service.dbm.datasources.

```
jonas.service.dbm.datasources            InstantDB1
```

InstantDB doit avoir été installé et configuré correctement, en utilisant une version supérieure à 3.14. (quelques exemples ont été testés avec une version 3.25).

Les classes du driver JDBC doivent être accessibles depuis le chemin d'accès classpath. Pour ceci, mettre à jour le fichier *config_env* ; (`$JONAS_ROOT/bin/unix/config_env` sous Unix, ou `%JONAS_ROOT%\bin\nt\config_env.bat` sur Windows).

Pour créer le compte dans la base de données en utilisant l'outil fourni avec InstantDB :

```
cd examples/src/eb
. $JONAS_ROOT/bin/unix/config_env
java org.enhydra.instantdb.ScriptTool Account.idb
```

Configurer d'autres bases de données

Le même type de procédé peut être utilisé pour les autres bases de données.Un modèle de Datasource pour PostgreSQL et pour InterBase est fourni avec JOnAS. Bien que plusieurs autres bases de données soient couramment mises en oeuvre par les utilisateurs de JOnAS (p. ex. Informix, Sybase, SQL Server), tous les drivers JDBC n'ont pas été testés avec JOnAS.

Configurer le Service de Sécurité

Une des propriétés du fichier `jonas.properties` permet de configurer le service de sécurité : la propriété `jonas.security.propagation` doit être positionnée à `true` (qui est la valeur par défaut), pour permettre la propagation du contexte de sécurité au travers des appels de méthode. Se référer également à la section "Propagation des contextes transactionnel et de sécurité" . Toute autre configuration de sécurité propre à JOnAS est faite dans le fichier jonas−realm.xml, et la configuration de la sécurité propre aux conteneurs web, certificats, etc., est faite dans les fichiers appropriés, ainsi que spécifié dans la section "Configurer la Sécurité".

Configurer le Service JMS

JOnAS utilise un produit d'origine tierce; actuellement le produit opensource Joram est intégré et livré avec JOnAS. Les produits SwiftMQ, WebSphere MQ d' IBM ainsi que des implémentations JMS d'autres fournisseurs peuvent facilement être intégrées. Le service JMS est utilisé pour contacter (ou charger) le MOM (*Message Oriented Middleware*) ou le *serveur JMS* correspondant. Les objets administrés par JMS utilisés par les composants EJB, telles les 'connection factories' et les destinations, doivent être créés avant l'exécution de l'EJB, en utilisant les possibilités d'administration propres à l'implémentation JMS. JOnAS fournit des "Wrappers" sur de telles APIs d'administration JMS, permettant au serveur EJB lui−même d'effectuer des opérations d'administration basiques.

Le *service jms* est un service optionnel, qui doit être lancé avant le *service conteneur ejb*.
Voici les propriétés qui peuvent être positionnées dans le fichier `jonas.properties` pour le *service jms* : `jonas.service.jms.collocated` pour positionner le mode de lancement du *Serveur JMS* . Lorsque la valeur positionnée est *true*, il est chargé dans la même JVM que le serveur JOnAS (c'est la valeur par défaut). Lorsqu'elle est positionnée à *false*, il est chargé dans une JVM séparée, et la propriété `jonas.service.jms.url` doit être positionnée à l'url permettant de se connecter sur le *serveur JMS*.

`jonas.service.ejb.mdbthreadpoolsize` est utilisé pour positionner le pool par défaut utilisé par les beans "Message driven" (Valeur par défaut : 10).

`jonas.service.jms.queues` et `jonas.service.jms.topics` sont utilisés pour positionner lors du chargement des listes de "queues" ou de "topics".

`jonas.service.jms.mom` indique quelle classe doit être utilisée pour exécuter les opérations d'administration. Cette classe est le "wrapper" vers l'implémentation du fournisseur JMS. La classe par défaut est *org.objectweb.jonas_jms.JmsAdminForJoram*, nécessaire pour Joram. En cas d'utilisation du produit SwiftMQ, vous pouvez récupérer une classe *com.swiftmq.appserver.jonas.JmsAdminForSwiftMQ* depuis le site SwiftMQ. Pour WebSphere MQ, la classe à utiliser est *org.objectweb.jonas_jms.JmsAdminForWSMQ*.

Des informations complémentaires sur la configuration JMS (en particulier, plusieurs aspects de la configuration avancée de JORAM) sont fournies dans les sections " Administration JMS" et "Utiliser un EJB exécutant des opérations JMS " du chapitre Utiliser JMS dans les composants applicatifs.

Des informations relatives à l'utilisation de WebSphere MQ sont fournies dans le manuel Utiliser WebSphere MQ JMS.

Configurer le Service Ressource

Le *Service Ressource* est un service optionnel qui doit être lancé dès lors que les composants EJB nécessitent un accès à un Système d'Information d'Entreprise (EIS: Enterprise Information Systems). La façon standard de procéder est d'utiliser un composant logiciel externe appelé *Connecteur*.

Le rôle du *Service Ressource* est de déployer les *Connecteurs* dans le serveur JOnAS, c.à.d. de les configurer dans un environnement opérationnel et d'enregistrer dans l'espace de nom JNDI une instance de *connection factory* qui pourra être accédée par les composants EJB.

Le *Service Ressource* peut être configuré d'une des façons suivantes :

- Le nom du ficher `rar` correspondant est listé dans la propriété `jonas.service.resource.resources` du fichier `jonas.properties`. Si le nom de fichier ne contient pas un nom de path absolu, alors il doit être positionné dans le répertoire **`$JONAS_BASE/rars/`**. Par exemple :

  ```
  jonas.service.resource.resources  MyEIS
  ```

  Ce fichier sera recherché dans le répertoire **`$JONAS_BASE/rars/`** . Cette propriété contient une liste de noms de fichiers *Connecteurs* séparés par des virgules (le suffixe '.rar' est optionnel).
- Une autre façon de déployer des fichiers *connecteurs* au démarrage du serveur est de placer les fichiers `rar` dans un répertoire *autoload*. Le nom du répertoire est spécifié en utilisant la propriété `jonas.service.resource.autoloaddir` du fichier `jonas.properties`. Ce répertoire est précisé par rapport au répertoire **`$JONAS_BASE/rars/`**.

Un fichier xml de *configuration des connecteurs spécifique à JOnAS* doit être inclus dans chaque connecteur. Ce fichier duplique les valeurs de toutes les propriétés de configuration déclarées dans le descripteur de déploiement du connecteur. Se référerer au chapitre Definir le descripteur de déploiement de connecteur JOnAS pour plus d'information.

Configurer le Service JMX

Le *Service JMX* doit être lancé pour administrer ou intrumenter le serveur JOnAS à l'aide de `JonasAdmin`. Ce service est obligatoire et sera lancé même s'il n'est pas présent dans la liste des services. Il se configure en choisissant une des deux implémentations JMX : SUN RI ou MX4J. Ce choix se fait selon la valeur de la propriété `jonas.service.jmx.class` du fichier de configuration JOnAS. Les deux valeurs possibles sont :

- `org.objectweb.jonas.jmx.sunri.JmxServiceImpl`, pour SUN RI
- `org.objectweb.jonas.jmx.mx4j.Mx4jJmxServiceImpl`, pour MX4J

Configurer le Service Mail

Le *Service Mail* est un service optionnel qui peut être utilisé pour l'envoi de courrier électronique (mail).
Il est basé sur JavaMail™ et sur l'API JavaBeans™ Activation Framework (JAF).

Une mail factory est nécessaire pour envoyer et recevoir du courrier électronique. JOnAS fournit deux types de mail factories : `javax.mail.Session` et `javax.mail.internet.MimePartDataSource`.
Cette dernière permet au mail d'être envoyé avec les champs sujet et destinataires pré−configurés.

Les objets d'une Mail factory doivent être configurés en tenant compte de leur type. Les sous−sections qui suivent décrivent brièvement comment doivent être configurés les objets `Session` et `MimePartDataSource`, afin d'exécuter les Bean session `SessionMailer` et `MimePartDSMailer` livrés avec la plate−forme.

Configurer la MailFactory Session

Le fichier modèle MailSession1.properties fourni dans le répertoire d'installation définit une mail factory de type `Session`. Le nom JNDI de l'objet factory est `mailSession_1`. Ce modèle doit être adapté avec les valeurs correspondant à votre installation.
Voir la section "Configurer une mail factory" ci−dessous pour la liste des propriétés disponibles.

Configurer la MailFactory MimePartDataSource

Le fichier modèle MailMimePartDS1.properties fourni dans le répertoire d'installation définit une mail factory de type `MimePartDSMailer` . Le nom JNDI de l'objet mail factory est `mailMimePartDS_1`. Ce modèle doit être adapté avec les valeurs correspondant à votre installation
Voir la section "Configurer une mail factory" pour la liste des propriétés disponibles.

La sous−section suivante fourni des informations sur la configuration de JOnAS lorsque l'on veut créer les objets mail factory nécessaires.

Configurer JOnAS

Les objets Mail factory crées par JOnAS doivent être nommés. Dans l'exemple mailsb, deux factories appelées *MailSession1* et *MailMimePartDS1* sont définies.
Chaque factory doit avoir un fichier de configuration dont le nom est le nom de la factory, avec l'extension `.properties` (`MailSession1.properties` pour la factory MailSession1).
De plus, le fichier `jonas.properties` doit définir la propriété `jonas.service.mail.factories`. Pour cet exemple, il s'agit de :

```
jonas.service.mail.factories MailSession1,MailMimePartDS1
```

Configurer une mail factory

Les champs sont les suivants :

Propriétés requises	
mail.factory.name	nom JNDI de la mail factory
mail.factory.type	Type de la factory. Cette propriété peut être `javax.mail.Session` ou `javax.mail.internet.MimePartDataSource`.
Propriétés optionnelles	
Propriétés d'Authentification	
mail.authentication.username	Positionne le nom utilisateur pour l'authentification.
mail.authentication.password	Définit le mot de passe pour l'authentification.
javax.mail.Session.properties (Se référer à JavaMail documentation pour plus d'information)	
mail.authentication.username	Positionne le nom utilisateur pour l'authentification.
mail.authentication.password	Définit le mot de passe pour l'authentification.
mail.debug	Mode debug initial. Par défaut : false.
mail.from	Adresse mail retour de l'utilisateur courant, utilisé par `InternetAddress` méthode `getLocalAddress.`
mail.mime.address.strict	La classe MimeMessage utilise `InternetAddress` méthode `parseHeader` pour analyser les en−têtes dans les messages. Cette propriété contrôle le flag 'strict' passé à la méthode `parseHeader`. Par défaut : true.
mail.host	Le nom par défaut du serveur mail utilisé conjointement pour le stockage et le transport. Utilisé si la propriété `mail.`*`protocol`*`.host` n'est pas positionnée.
mail.store.protocol	Spécifie le protocole par défaut d'accès aux messages. `La méthode Session getStore()` retourne un objet Store qui implémente ce protocole. Par défaut, le premier provider Store trouvé dans les fichiers de configuration est retourné.
mail.transport.protocol	Spécifie le protocole par défaut d'accès aux messages. La méthode `Session getTransport()` retourne un objet Transport qui implémente ce protocole. Par défaut, le premier fournisseur de Transport trouvé dans les fichiers de configuration est retourné.
mail.user	Le nom utilisateur à utiliser par défaut lors de la connexion au serveur mail. Utilisé si la propriété `mail.`*`protocol`*`.user` n'est pas positionnée.
mail.<protocol>.class	Spécifie le nom de classe pleinement qualifié du provider pour le protocole spécifié. Utilisé lorsqu'il existe plus d'un provider pour un protocole donné ; Cette propriété peut être utilisée pour spécifier le provider par défaut. Le provider doit toujours être listé dans un fichier de configuration.

mail.<protocol>.host	Le nom de système du serveur mail pour le protocole spécifié. Ecrase la propriété `mail.host`.
mail.<protocol>.port	Le numéro de port du serveur mail pour le protocole spécifié. S'il n'est pas précisé, le port par défaut du protocole est utilisé.
mail.<protocol>.user	Le nom utilisateur à positionner lors d'une connexion à des serveurs mail utilisant le protocole spécifié. Ecrase la propriété `mail.user` .
MimePartDataSource properties (**Utilisé uniquement si mail.factory.type est positionné à javax.mail.internet.MimePartDataSource**)	
mail.to	Positionne la liste des premiers destinataires ("to") du message.
mail.cc	Positionne la liste des destinataires du message en Copie Carbonne ("cc").
mail.bcc	Positionne la liste des destinataires du message en Copie Carbonne Cachée (Blind Carbon Copy) ("bcc") .
mail.subject	Positionne le sujet du message.

Configurer la Sécurité

Configurer la sécurité JAAS pour l'authentification par certificat

Le *Service Sécurité* est utilisé par le *Service Conteneur* pour fournir la sécurité des composants EJB. Le *Service Conteneur* fournit deux formes de sécurité : sécurité déclarative, et sécurité programmable. Le *Service Sécurité* exploite *les rôles de sécurité* et les *permissions méthodes* localisées dans le descripteur de déploiement.

Notez que :

- JOnAS s'appuie sur Tomcat ou Jetty pour l'identification des clients web. Les clients java utilisent les modules de login JAAS pour l'identification. JOnAS prend en charge l'authentification de l'utilisateur.
  JOnAS fournit trois types de Realm, qui sont définis dans le fichier `$JONAS_ROOT/conf/jonas-realm.xml`:
  - Memory realm: utilisateurs, groupes et rôles sont écrits dans la section `<jonas-memoryrealm>` du fichier.
  - Datasource realm: Les informations sur les utilisateurs/rôles/groupes sont stockées dans une base de données. La configuration pour accéder à la source de donnée correspondante est décrite dans la section `<jonas-dsrealm>` du fichier `$JONAS_ROOT/conf/jonas-realm.xml`.
    La configuration requiert le nom de la source de donnée, les tables utilisées et le nom des colonnes.
  - LDAP realm: Les informations sur les utilisateurs/rôles/groupes sont stockées dans u annuaire LDAP. Ceci est décrit dans la section `<jonas-ldaprealm>` du fichier `$JONAS_ROOT/conf/jonas-realm.xml`.
    Il y a des paramètres optionnels. Si certains d'entre eux ne sont pas spécifiés, ils sont positionnés à des valeurs par défaut.

    p. ex. : si l'élément `providerUrl` n'est pas positionné, la valeur par défaut est

`ldap://localhost:389.`
Editez le fichier `jonas-realm_1_0.dtd` DTD pour voir les valeurs par défaut.
Pour Tomcat, utilisez le realm `org.objectweb.jonas.security.realm.JRealmCatalina41.`
Pour Jetty, utilisez le realm `org.objectweb.jonas.security.realm.JRealmJetty42.`
Ces realms nécessitent en argument le **nom** de la ressource utilisée pour l'authentification. C'est le nom de ressource qui figure dans le fichier `jonas-realm.xml`.

- Il n'y a pas de correspondance pour la sécurité entre JOnAS et l'environnement opérationnel cible. Plus précisément, les rôles définis pour JOnAS ne trouvent pas leur correspondance dans l'environnement opérationnel (p. ex., les groupes Unix).

Une propriété du fichier `jonas.properties` permet de configurer le service de sécurité : la propriété `jonas.security.propagation` doit être positionnée à `true` (ce qui est la valeur par défaut), pour permettre la propagation du contexte de sécurité au travers des appels méthodes. Se référer également à la section "Propagation des contextes de sécurité et de transaction" .

Utiliser les intercepteurs du conteneur Web Tomcat 4.1.x pour l'authentification

Ceci se fait avec Tomcat 4.1.x, en remplaçant dans le fichier `$JONAS_ROOT/conf/server.xml`,
ou dans le fichier `$JONAS_BASE/conf/server.xml`,
ou dans le fichier `$CATALINA_HOME/conf/server.xml`,
ou `dans $CATALINA_BASE/conf/server.xml`

la ligne :

```
<Realm className="org.apache.catalina.realm.UserDatabaseRealm" debug="0" resourceName="UserDatabase" /
```

Par la ligne:

```
<Realm className="org.objectweb.jonas.security.realm.JRealmCatalina41" debug="0" resourceName="memrlm_
```

Un ressource mémoire, source de données ou LDAP peut être utilisée pour l'authentification, en positionnant resourceName avec le nom correct de la ressource spécifiée, c'est-à-dire : memrlm_1, memrlm_2, dsrlm_1, ldaprlm_1, etc.

Utiliser les intercepteurs Jetty pour l'authentification

Pour utiliser les intercepteurs Jetty dans une application web, il faut fournir un fichier `web-jetty.xml` sous le répertoire `WEB-INF` dans l'archive `.war`, dans lequel il est spécifié que l'intercepteur à utiliser est `org.objectweb.jonas.security.realm.JRealmJetty42` for JOnAS et non celui par défaut. Ceci est illustré dans l'exemple `earsample` :

```
<Call name="setRealmName">
  <Arg>Example Basic Authentication Area</Arg>
</Call>
```

```
<Call name="setRealm">
  <Arg>
    <New class="org.objectweb.jonas.security.realm.JRealmJetty42">
      <Arg>Example Basic Authentication Area</Arg>
      <Arg>memrlm_1</Arg>
    </New>
  </Arg>
</Call>
```

Différents fichiers web−jetty.xml sont présentés dans l'exemple `earsample` et la démonstration `alarm`.

Configurer la correspondance principal/rôles

JOnAS s'appuie sur le fichier `jonas−realm.xml` pour le contrôle d'accès aux méthodes des composants EJB. Exemple de bean sécurisé avec le rôle jonas:

```
<assembly−descriptor>
  <security−role>
    <role−name>jonas</role−name>
  </security−role>

  <method−permission>
    <role−name>jonas</role−name>
    <method>
          <ejb−name>Bean</ejb−name>
          <method−name>*</method−name>
    </method>
  </method−permission>
  ...
  ...
</assembly−descriptor>
```

La sous−section suivante décrit comment configurer les différentes ressources liées à l'authentification lorsqu'il est nécessaire d'ajouter un utilisateur avec le rôle spécifié (jonas) et autorisé à appeler des méthodes, etc, ...

Configurer les ressources mémoire dans le fichier `jonas−realm.xml`

Pour ajouter le rôle '`jonas`', placer les lignes suivantes dans la section `<roles>` :

```
<roles>
  <role name="jonas" description="Role used in the sample security howto" />
</roles>
```

Ensuite, pour ajouter un utilisateur avec ce rôle spécifique :
Ajouter un utilisateur avec le rôle '`jonas`' role dans la section `<users>` :

```
<users>
```

```
    <user name="jonas_user" password="jonas_password" roles="jonas" />
  </users>
```

La section `<groups>` permet de grouper les rôles.

Exemple : Un développeur réalise deux applications, une nécessitant le rôle role1 et la seconde application nécessitant le rôle role2. Plutôt que de déclarer deux utilisateurs, user1 avec le rôle role1 et user2 avec le rôle role2, il est possible de déclarer un groupe. Ce groupe va avoir deux rôles : role1 and role2.

```
<groups>
  <group name="myApplications" roles="role1,role2" description="Roles for my applications" />
</groups>
```

Un utilisateur de nom 'john' peut être ajouté au groupe 'myApplications'. Cet utilisateur peut être ajouté dans la section `<users>...</users>` de la section `<memoryrealm name="...">...</memoryrealm>` du fichier `jonas-realm.xml`.

```
<user name="john" password="john_password" groups="myApplications" />
```

Cet utilisateur aura deux rôles, role1 et role2 puisque ces rôles sont dans le groupe appelé 'myApplications'.
Un nouvel utilisateur 'joe' peut être déclaré avec le même groupe.

```
<user name="joe" password="joe_password" groups="myApplications" />
```

Bien sur, l'utilisateur joe peut être dans un autre groupe, par exemple le groupe 'otherApplications'.

```
<user name="joe" password="joe_password" groups="myApplications,otherApplications" />
```

Si joe doit pouvoir s'authentifier sur l'application web jonasAdmin, il nécessite le rôle 'jonas'

```
<user name="joe" password="joe_password" groups="myApplications,otherApplications" roles="jonas" />
```

Ainsi, un utilisateur peut avoir plusieurs groupes et plusieurs rôles définis.

Maintenant, si le développeur ajoute une application tierce qui nécessite le rôle 'role3', le développeur a juste besoin d'ajouter ce rôle dans le groupe "myApplications' et tous les utilisateurs du groupe 'myApplications' auront ce rôle.

```
<groups>
  <group name="myApplications" roles="role1,role2,role3" description="Roles for my applications" />
</groups>
```

Ajouter la ressource mémoire dans le fichier `jonas-realm.xml` :
Notez : [...] signifie qu'il peut y avoir plusieurs memory-realms puisque plusieurs memory-realm peuvent être

déclarés dans la section <jonas−memoryrealm>
Un memory−realm peut être utilisé pour Tomcat, un autre pour Jetty, ou d'autres encore pour des contextes spécifiques (dans server.xml pour Tomcat ou web−jetty.xml, jetty.xml pour Jetty).

```
<jonas−memoryrealm>
  [...]
  <memoryrealm name="howto_memory_1">
    <roles>
      <role name="jonas" description="Role used in the sample security howto" />
    </roles>

    <users>
      <user name="jonas_user" password="jonas_password" roles="jonas" />
    </users>
  </memoryrealm>
  [...]
</jonas−memoryrealm>
```

Notez qu'il est mieux d'utiliser l'application web jonasAdmin pour ajouter/détruire des utilisateurs, groupes, rôles plutôt qu'éditer le fichier jonas−realm.xml manuellement. Certaines erreurs de configuration peuvent ainsi être évitées.

Configurer les ressources Datasource dans le fichier `jonas−realm.xml`

Tout d'abord, construire les tables dans lesquelles les utilisateurs et rôles vont être stockés.
Exemple de tables :
realm_users : Ajouter un utilisateur `jonas_user` avec le mot de passe `jonas_password`

user_name	user_pass
...	...
jonas_user	jonas_password
...	...

Notez que la table peut contenir plus de deux colonnes.

realm_roles : Ajouter le rôle `jonas` à l'utilisateur `jonas_user`

user_name	role_name
...	...
jonas_user	jonas

... ...

Maintenant, declarez la ressource dans le fichier `jonas-realm.xml`.
L'élément `dsName` décrit le nom JNDI de la source de donnée à utiliser.
Ensuite, une configuration de la source de donnée avec le nom JNDI correct pour le service `dbm` doit être positionnée dans le fichier `jonas.properties`.
La ressource source de données à ajouter dans le fichier `jonas-realm.xml` est :

```
<jonas-dsrealm>
  [...]
  <dsrealm name="howto_datasource_realm1"
           dsName="jdbc_1"
           userTable="realm_users" userTableUsernameCol="user_name" userTablePasswordCol="user_pass"
           roleTable="realm_roles" roleTableUsernameCol="user_name" roleTableRolenameCol="role_name"
         />
  [...]
</jonas-dsrealm>
```

Configurer les ressources LDAP dans le fichier `jonas-realm.xml`

L'utilisateur est ajouté dans le serveur LDAP.
Dans ce cas, tous les utilisateurs sont sur le DN `ou=people,dc=jonas,dc=objectweb,dc=org`.
Par exemple, le nom unique pour l'utilisateur `'jonas_user'` sera : DN `uid=jonas_user,ou=people,dc=jonas,dc=objectweb,dc=org` .
Le rôle jonas sera ajouté au DN `ou=groups,dc=jonas,dc=objectweb,dc=org`.
dans ce cas : DN `cn=jaas,ou=groups,dc=jonas,dc=objectweb,dc=org`
L'utilisateur est ajouté au rôle en ajoutant un champ 'uniquemember' au rôle. `uniquemember = uid=jonas,ou=people,dc=jonas,dc=objectweb,dc=org`
Format LDIF pour l'utilisateur :

```
# jonas_user, people, jonas, objectweb, org
dn: uid=jonas_user,ou=people,dc=jonas,dc=objectweb,dc=org
objectClass: inetOrgPerson
uid: jonas_user
sn: jonas_user
cn: JOnAS user
userPassword:: jonas_password
```

Format LDIF pour le rôle :

```
# jonas, groups, jonas, objectweb, org
dn: cn=jonas,ou=groups,dc=jonas,dc=objectweb,dc=org
objectClass: groupOfUniqueNames
uniqueMember: uid=jonas_user,ou=people,dc=jonas,dc=objectweb,dc=org
cn: jonas
```

Maintenant, le fichier `jonas-realm.xml` peut être personnalisé en ajoutant une ressource LDAP.
Il existe deux méthodes pour l'authentification : la méthode bind ou la méthode comparative.

- La méthode bind : Pour vérifier les droits d'accès, la ressource essaie de se connecter au serveur LDAP avec le nom d'utilisateur et le mot de passe donnés.
- La méthode comparative : La ressource récupère le mot de passe de l'utilisateur depuis le serveur LDAP, et compare ce mot de passe à celui donné par l'utilisateur. *Notez que cette méthode requiert les rôles d'admin dans la configuration, de façon à pouvoir lire les mots de passe des utilisateurs.*.

Par défaut, la méthode bind est utilisée.
Tous les éléments de configuration pour la ressource ldap peuvent être trouvés dans le fichier DTD jonas-realm_1_0.dtd.

Pour cet exemple, on considère que le serveur LDAP est sur le même système, et qu'il est accessible sur le port par défaut (389). Il prend toutes les valeurs par défaut du DTD.
La ressource source de donnée à ajouter dans le fichier `jonas-realm.xml` est :

```
<jonas-ldaprealm>
  [...]
  <ldaprealm name="howto_ldap_realm1"
             baseDN="dc=jonas,dc=objectweb,dc=org" />
  [...]
</jonas-ldaprealm>
```

Configurer l'authentification basée sur un certificat client dans le conteneur web

1. Introduction

Pour installer l'authentification du client, basée sur la présence du certificat client dans un conteneur web, suivre les étapes suivantes :

- Configurer le Realm que le conteneur Web devra utiliser.
- Configurer un listener SSL sur le conteneur Web.
- Configurer l'appliction Web pour qu'elle demande un certificat client.
- Configurer les LoginModules du service JAAS
- Renseigner la liste d'accès du Realm .

Il est obligatoire de posséder un certificat X.509 pour votre conteneur Web sur chaque interface externe (adresse IP) qui accepte des connexions sécurisées. Celui-ci peut être signé de façon digitale par une Autorité de Certification, ou être auto-signé.

2. Configurer le Realm que le conteneur Web aura à utiliser

Si vous utilisez Tomcat

Si le package JOnAS−Tomcat est installé, passez ce paragraphe.

Avec Tomcat 4.1.x, dans les fichiers `$JONAS_ROOT/conf/server.xml`,
ou `$JONAS_BASE/conf/server.xml`,
ou `$CATALINA_HOME/conf/server.xml`,
ou `$CATALINA_BASE/conf/server.xml` remplacez le Realm courant par le suivant :

```
<Realm className="org.objectweb.jonas.security.realm.JRealmJAASCatalina41" />
```

La classe spécifiée utilise le modèle JAAS pour authentifier les utilisateurs. Ainsi, pour choisir la ressource dans laquelle rechercher l'authentification des données, configurer JAAS.

Si vous utilisez Jetty

Editer le fichier `web−jetty.xml` sous le répertoire `WEB−INF` dans le fichier `.war` pour déclarer le nom du Realm et le Realm:

```
<Configure class="org.mortbay.jetty.servlet.WebApplicationContext">
...
!−− Set the same realm name as the one specified in <realm−name> in <login−config> in the web.xml file
<Call name="setRealmName">
  <Arg>Example Authentication Area</Arg>
</Call>
<!−− Set the class Jetty has to use to authenticate the user and a title name for the pop−up window −−
<Call name="setRealm">
  <Arg>
    <New class="org.objectweb.jonas.security.realm.JRealmJAASJetty42">
      <Arg>JAAS on Jetty</Arg>
    </New>
  </Arg>
</Call>
...
</Configure>
```

La classe spécifiée utilise le model JAAS pour authentifier l'utilisateur. Ainsi, pour choisir la ressource dans laquelle rechercher l'authentification des données, configurer JAAS.

3. Configurer un listener SSL sur le conteneur Web

Pour Tomcat

Enlever la marque de commentaire devant la section suivante du fichier `server.xml` :

```
  <Connector className="org.apache.catalina.connector.http.HttpConnector" port="8443"
             minProcessors="5" maxProcessors="75" enableLookups="true" acceptCount="10"
```

```
            debug="0" scheme="https" secure="true">
    <Factory className="org.apache.catalina.net.SSLServerSocketFactory" clientAuth="false" protocol="T
  </Connector>
```

Important : Positionner le paramètre `clientAuth` à faux, sinon toutes les applications Web nécessitant SSL vont demander un certificat client. L'authentification du client sera configurée ultérieurement dans le fichier `web.xml` des archives `.war` spécifiques.

Pour plus d'information, se référer à http://jakarta.apache.org/tomcat/tomcat−4.1−doc/ssl−howto.html.

Pour Jetty

Dans le descripteur global de déploiement de Jetty (le fichier `jetty.xml`), localisé dans le répertoire `$JETTY_HOME/conf` ou `$JONAS_BASE/conf` (suivant que Jetty est utilisé en mode autonome ou non), positionnez :

```
<Configure class="org.mortbay.jetty.Server">
...
<Call name="addListener">
<Arg>
        <New class="org.mortbay.http.SunJsseListener">
                <Set name="Port">8443</Set>
                <Set name="Keystore">
                        <SystemProperty name="jonas.base" default="."/>/conf/keystore</Set>
                <Set name="Password">changeit</Set>
                <Set name="KeyPassword">jettykeypassword</Set>
                <Set name="NeedClientAuth">true</Set>
        </New>
</Arg>
</Call>
...
</configure>
```

Le certificat Jetty est stocké dans un dépot nommé "keystore". C'est le certificat que va présenter Jetty lorsqu'une connexion est sur le point d'être établie.

Pour plus d'information, se référer à http://jetty.mortbay.org/jetty/doc/SslListener.html.

4. Configurer votre application Web pour qu'elle demande un certificat client

Ajouter les lignes suivantes au fichier `web.xml` de l'archive `.war` de l'application web.

```
<login−config>
        <auth−method>CLIENT−CERT</auth−method>
        <realm−name>Example Authentication Area</realm−name>
```

```
</login-config>
```

Important:

- Que ce soit pour Tomcat ou Jetty, vérifier que la zone d'accès restreint de l'application Web est configurée dans le fichier `web.xml` de l'archive `.war` avec une déclaration de `security-constraint`.
- Que ce soit pour Tomcat ou Jetty, vérifier que l'application web est toujours accédée par SSL, sans quoi le conteneur web ne récupérera pas de certificat et ne sera pas capable d'authentifier l'utilisateur.
- Que ce soit pour Tomcat ou Jetty, lorsque l'authentification par certificat client est activée, le navigateur web de l'utilisateur reçoit la liste des Autorités de Certification de Confiance (de votre application web). Une connexion sera établie avec le client uniquement s'il présente un certificat issu d'une des Authorités de Certification de Confiance, elle sera rejetée dans le cas contraire.
    - Dans Jetty, les certificats de toutes les Autorités de Certification de Confiance doivent être importées dans le "keystore" spécifié par le listener SSL.
    - Dans Tomcat, les certificats de toutes les Autorités de Certification de Confiance doivent être importées dans le fichier "keystore" `$JAVA_HOME/jre/lib/security/cacerts` (Personnaliser le listener SSL pour modifier cette localisation).
- Pour Jetty, si le paramètre `NeedClientAuth` du fichier `jetty.xml` est positionné à true, toutes les applications Web accédées par SSL demanderont à l'utilisateur de présenter un certificat. Utilisez différents listeners SSL sur différents ports pour avoir simultanément des applications Web demandant un certificat client, et d'autres applications n'en demandant pas.
- Pour Jetty, pensez à positionner la même propriété `realm-name` que celle du fichier `web-jetty.xml`.

Pour Jetty, se référer à http://jetty.mortbay.org/jetty/doc/ClientCert.html pour plus d'information.

5. Configurer les LoginModules JAAS

Ce processus d'authentification est construit sur la technologie JAAS. Ceci signifie que l'authentification est pluggable, et que la technologie d'authentification voulue est spécifiée lors de l'exécution. Vous devez donc choisir les `LoginModule`s à utiliser dans votre conteneur Web pour authentifier les utilisateurs.

choisir les LoginModules

```
"org.objectweb.jonas.security.auth.spi.JResourceLoginModule"
```

C'est le `LoginModule` principal. Il est fortement recommandé de l'utiliser dans toute authentification, du fait qu'il vérifie l'information d'authentification de l'utilisateur dans la ressource spécifiée.
Il nécessite uniquement un paramètre :

- **`resourceName`**: le nom de l'entrée dans le fichier "jonas-realm.xml " utilisé; cette entrée représente

comment et où l'information d'authentification est stockée.
- **`certCallback`**: Spécifiez ce paramètre optionnel si vous voulez que ce module de connexion demande le callback du certificat. Par défaut, il est positionné à false. Il devrait être positionné à true lorsque l'on utilise des Realm JAAS avec certificats.

```
"org.objectweb.jonas.security.auth.spi.CRLLoginModule"
```

Le LoginModule contient l'authentification basée sur les certificats. Cependant, lorsqu'il est validé, il permet également des accès non basés sur les certificats. Il vérifie que le certificat présenté par l'utilisateur n'a pas été révoqué par l'Autorité de Certification qui l'a signé. Pour l'utiliser, il est nécessaire qu'existe soit le répertoire dans lequel stocker les fichiers de listes de révocation (CRLs) soit un annuaire LDAP.

Il peut prendre les paramètres suivants :

- **`CRLsResourceName`**: ce paramètre spécifie comment sont stockés les CRLs :
  - `"Directory"`: si les fichier CRL sont stockés dans un répertoire sur le système; un autre pointeur vers ce répertoire doit être spécifié :
    - **`CRLsDirectoryName`**: le répertoire contenant le fichier CRL (l'extention de ces fichiers doit être .crl).
  - `"LDAP"` : si vos fichiers CRL sont stockés dans un annuaire LDAP ; **cette fonctionnalité est expérimentale**; deux paramètres additionnels doivent être spécifiés:
    - **`address`**: l'adresse du serveur qui héberge l'annuaire LDAP
    - **`port`**: le port utilisé par l'annuaire LDAP ; Les CRLs sont extraits d'un annuaire LDAP en utilisant le schéma LDAP défini dans le RFC 2587.

Spécifier les paramètres de configuration

La configuration JAAS repose sur un fichier dans lequel le module login pour utiliser l'authentification est décrit. Ce fichier est localisé sous `$JONAS_BASE/conf` et est nommé `jaas.config`. Pour changer sa localisation et son nom, éditer le script `$JONAS_BASE/bin/jonas.sh` et modifier les lignes suivantes:

```
-Djava.security.auth.login.config=$JONAS_BASE/conf/jaas.config
```

Le fichier de configuration JAAS suit cette structure :

```
Application_1 {
  LoginModuleClassA Flag Options;
  LoginModuleClassB Flag Options;
  LoginModuleClassC Flag Options;
```

```
};

Application_2  {
  LoginModuleClassB Flag Options;
  LoginModuleClassC Flag Options;
};

Other {
  LoginModuleClassC Flag Options;
  LoginModuleClassA Flag Options;
};
```

Exemple de configuration avec un répertoire CRL :

```
tomcat {
  org.objectweb.jonas.security.auth.spi.CRLLoginModule required
  CRLsResourceName="Directory"
  CRLsDirectoryName="path_to/CRLs";

  org.objectweb.jonas.security.auth.spi.JResourceLoginModule
  required
  resourceName="memrlm_1";
};
```

Il peut y avoir de multiples entrées dans ce fichier, spécifiant différentes configurations que JOnAS peut utiliser. Il y a deux pré−requis : l'entrée dédiée à Tomcat doit être nommée `tomcat`, et l'entrée Jetty, `jetty`. Notez que tout dans ce fichier est sensible aux types des caractères (minuscules/majuscules).

On utilise un flag associé aux LoginModules pour configurer leur comportement en cas de succès ou d'échec:

- `required` − Le LoginModule doit réussir. S'il réussit ou échoue, l'authentification continue sur la suite de la liste du LoginModule.
- `requisite` − Le LoginModule doit réussir. S'il réussit, l'authentification continue sur la suite de la liste du LoginModule. S'il échoue, le contrôle retourne immédiatement à l'application (l'authentification ne se poursuit pas sur la suite de la liste du LoginModule).
- `sufficient` − Le LoginModule ne doit pas obligatoirement réussir. S'il réussit, le contrôle retourne immédiatement à l'application (l'authentification ne se poursuit pas sur la liste du LoginModule). S'il échoue, l'authentification continue sur la suite de la liste du LoginModule.
- `optimal` − Le LoginModule ne doit pas obligatoirement réussir. Qu'il aboutisse ou qu'il échoue, l'authentification continue sur la suite de la liste du LoginModule.

6. Renseigner la liste d'accès du Realm

Les utilisateurs n'auront plus à entrer un login/password. Ils auront uniquement à présenter leurs certificats et l'authentification sera effectuée de façon transparente par le navigateur (après que l'utilisateur y ait importé son certificat) . De ce fait, l'identité présentée à un serveur n'est pas un login, mais un "Distinguished Name" : c'est le nom identifiant la personne à laquelle appartient le certificat.

Ce nom ressemble à ce qui suit :

```
CN=Someone Unknown, OU=ObjectWeb, O=JOnAS, C=ORG
```

E : Email Address
CN : Common Name
OU : Organizational Unit
O : Organization
L : Locality
ST : State or Province Name
C : Country Name

Le `Subject` dans un certificat contient les attributs principaux, et peut en inclure des secondaires, comme : le Titre, le Nom de Rue, le Code postal, le numéro de téléphone.
Dans les versions précédentes, un realm memory du fichier `jonas-realm.xml` pouvait contenir :

```
<user name="jps_admin" password="admin" roles="administrator"/>
```

Pour spécifier un accès utilisateur basé sur un certificat, le champ DN associé à l'utilisateur précédé de la chaine ##DN## doit maintenant être entré, comme il est montré dans l'exemple suivant :

```
<user name="##DN##CN=whale, OU=ObjectWeb, O=JOnAS, L=JOnAS, ST=JOnAS, C=ORG" password="" roles="jadmi
```

Configurer les sources de données JDBC

Cette section décrit comment configurer les sources de données pour permettre l'accès aux bases de données depuis les applications.

Configurer les sources de données

Que la persistance soit gérée par les conteneurs ou par les beans, les systèmes de stockage utilisés par JOnAS au travers de son interface JDBC sont des systèmes relationnels. Les connexions JDBC sont obtenues au travers d'un objet, la **DataSource** (source de donnée), fourni par le serveur d'application. L'interface DataSource est définie dans les extensions du standard JDBC 2.0. Un objet DataSource identifie à la fois une base de données et un moyen d'y accéder au travers de JDBC (par un driver JDBC). Un serveur d'application peut accéder à plusieurs bases de données, et fournit alors les objets Datasource correspondants. Des objets DataSource disponibles peuvent être ajoutés à la

plate−forme; ils doivent être spécifiés dans le fichier `jonas.properties`. Cette section explique comment les objets Datasource peuvent être définis et configurés dans le serveur JOnAS.

Pour supporter les transactions distribuées, JOnAS requiert l'utilisation d'un driver conforme JDBC2−XA. De tels drivers, implémentant l'interface **XADataSource** ne sont pas toujours disponibles pour les bases de données relationnelles. JOnAS fournit un **driver−wrapper** générique, qui émule l'interface XADataSource sur un driver JDBC standard. Il est important de noter que ce driver−wrapper n'assure pas un réel commit à deux phases pour les transactions distribuées.

Le **driver−wrapper** de JOnAS fournit une implémentation de l'interface DataSource permettant de définir des objets DataSource en utilisant un driver conforme à JDBC1 pour certains produits de bases de données relationnelles, tels Oracle, PostGres, ou InstantDB.

Ni les spécifications EJB ni les spécifications J2EE ne décrivent comment définir les objets DataSource de façon à les rendre disponibles à une plate−forme J2EE. Ce document, qui décrit comment définir et configurer des objets DataSource, est donc *specifique à JOnAS*. Cependant, l'utilisation de ces objets DataSource dans les composants applicatifs est standard, à travers l'utilisation des références à des ressources (se référer à l'exemple de la section "Ecrire des opérations d'accès BD" du guide Développer des Entity Bean).

Un objet DataSource doit être défini dans le fichier nommé <DataSource name>.properties (par exemple Oracle1.properties pour une source de données Oracle et InstantDB1.properties pour une source de données InstantDB, sont délivrés avec la plate−forme).

Dans le fichier jonas.properties, pour définir une source de données "Oracle1" ajouter le nom "Oracle1" (nom du fichier de propriétés) à la ligne jonas.service.dbm.datasources, comme ceci :

```
jonas.service.dbm.datasources Oracle1,InstantDB1,PostgreSQL
```

Le fichier de propriétés définissant une source de données doit contenir l'information suivante :

datasource.name	Nom JNDI de la source de donnée
datasource.url	URL JDBC de la base de donnée : jdbc:<database_vendor_subprotocol>:...
datasource.classname	Nom de la classe implémentant le driver JDBC
datasource.username	Nom de l'utilisateur de la base de donnée
datasource.password	Mot de passe de l'utilisateur de la base de données

Un objet DataSource pour Oracle (par exemple, Oracle1), nommé *jdbc_1* dans JNDI, et utilisant le driver JDBC *thin* d'Oracle, doit être décrit dans un fichier appelé `Oracle1.properties`, comme dans l'exemple suivant :

```
datasource.name       jdbc_1
datasource.url        jdbc:oracle:thin:@malte:1521:ORA1
datasource.classname  oracle.jdbc.driver.OracleDriver
datasource.username   scott
datasource.password   tiger
```

Dans cet exemple, "malte" est le nom système du serveur sur lequel s'exécute Oracle DBMS, 1521 est le numéro de port SQL*Net V2 sur ce serveur, et ORA1 est l' ORACLE_SID.

Cet exemple utilise le driver JDBC "Thin" d'Oracle. Si votre serveur d'application tourne sur le même système que le serveur Oracle (Oracle DBMS), vous pouvez utiliser le driver JDBC OCI d'Oracle ; suivant la version d'Oracle, l'URL à utiliser sera jdbc:oracle:oci7, ou jdbc:oracle:oci8. Les drivers JDBC d'Oracle peuvent être téléchargés sur le site web d' Oracle.

Pour créer un objet source de donnée InstantDB (par exemple, InstantDB1) nommé *jdbc_2* dans JNDI, entrez les lignes suivantes (dans un fichier `InstantDB1.properties`) :

```
datasource.name       jdbc_2
datasource.url        jdbc:idb=db1.prp
datasource.classname  jdbc.idbDriver
datasource.username   useless
datasource.password   useless
```

Pour créer un objet source de donnée PostGreSQL nommé *jdbc_3* in JNDI, entrez les lignes suivantes (dans un fichier `PostGreSQL.properties`):

```
datasource.name       jdbc_3
datasource.url        jdbc:postgresql://your_host/your_db
datasource.classname  org.postgresql.Driver
datasource.username   useless
datasource.password   useless
```

Les propriétés ayant la valeur "useless" ne sont pas utilisées pour ce type de stockage de la persistance.

Le nom et le mot de passe de l'utilisateur de la base de données peuvent soit être placés dans la description du DataSource (<DataSource name>.properties file) et permettre aux composants applicatifs d'utiliser la méthode getConnection(), soit ne pas y être placés et permettre aux composants applicatifs d'utiliser la méthode getConnection(String username, String password). Dans la référence sur ressource de la Datasource dans le descripteur de déploiement, l'élément <res−auth> doit avoir la valeur correspondante, i. e. 'Container' ou 'Application'.

CMP2.0/JORM

Pour implémenter la persistance EJB 2.0 (CMP2.0), JOnAS repose sur le framework JORM. JORM doit adapter sa correspondance Objet/Relationnel à la base sur laquelle il s'appuie et utilise pour cela des adapteurs appelés "mappers". Pour chaque type de base de données (et plus précisémment pour chaque driver JDBC), le mapper correspondant doit être spécifié dans la Datasource. C'est l'objet de la propriété ***datasource.mapper***.

property name	description	possible values
datasource.mapper	JORM database mapper	• rdb: mapper générique (driver JDBC standard ...)

		• rdb.postgres: mapper pour PostgreSQL • rdb.oracle8: mapper pour Oracle 8 et versions antérieures • rdb.oracle: mapper pour Oracle 9 • rdb.mckoi: mapper pour McKoi Db • rdb.mysql: mapper pour MySQL

Configuration du ConnectionManager

Chaque Datasource est implémenté comme gestionnaire de connexions pouvant être configuré à l'aide de propriétés additionnelles décrites dans la table qui suit. Se référer au fichier Oracle1.properties pour exemple. Ces propriétés sont toutes optionnelles, et possèdent des valeurs par défaut.

property name	description	default value
jdbc.conncheckleve l	Niveau de vérification des connexions JDBC	0 (pas de vérification), 1 (Vérifier la connexion), higher (exécuter la requête de test)
jdbc.connmaxage	age max pour les connexions jdbc	30
jdbc.conntestst mt	requête de test	select 1
jdbc.minconpool	Nombre minimum de connexions dans le pool	0
jdbc.maxconpool	Nombre maximum de connexions dans le pool	−1 (i.e. pas de limite supérieure)

jdbc.conntestst mt n'est pas utilisé lorsque jdbc.conncheckleve l est égal à 0 ou 1.

jdbc.minconpool est utilisé lors de la création de la DataSource, aucune modification n'est prise en compte pour une DataSource déjà créée.

jdbc.maxconpool peut être augmenté ou diminué dynamiquement.

Tracer les requêtes SQL avec P6Spy

L'outil P6Spy est intégré à JOnAS pour fournir un environnement de trace des requêtes SQL envoyées à la base de donnée.

Pour valider cette fonctionnalité de trace, respecter les étapes suivantes :

- positionner la propriété `datasource.classname` de votre fichier propriété des sources de données à `com.p6spy.engine.spy.P6SpyDriver`
- positionner la propriété `realdriver` du fichier `spy.properties` (localisé sous $JONAS_BASE/conf) au driver jdbc de votre base de données actuelle
- vérifier que `logger.org.objectweb.jonas.jdbc.sql.level`est positionné à DEBUG dans $JONAS_BASE/conf/trace.properties.

Exemple:

Contenu du fichier Datasource properties:

```
datasource.name       jdbc_3
datasource.url        jdbc:postgresql://your_host:port/your_db
datasource.classname  com.p6spy.engine.spy.P6SpyDriver
datasource.username   jonas
datasource.password   jonas
datasource.mapper     rdb.postgres
```

Dans le fichier JONAS_BASE/conf/spy.properties :

```
realdriver=org.postgresql.Driver
```

Dans le fichier JONAS_BASE/conf/trace.properties:

```
logger.org.objectweb.jonas.jdbc.sql.level  DEBUG
```

Configurer les Connecteurs JDBC

Plutôt que d'utiliser le service "base de données" pour configurer des sources de données JDBC, comme décrit dans la section précédente, il est aussi possible d'utiliser le service "ressource" et des connecteurs JDBC conformes à la spécification d'architecture connecteur de J2EE. La fonction obtenue est la même, est l'on bénéficie en plus d'une gestion de "pools" de requêtes JDBC pré−compilées. Cette section décrit la façon dont doivent être configurés les connecteurs JDBC pour connecter les applications aux bases de données.

Configurer les Connecteurs

Que la persistance soit gérée par les conteneurs ou par les beans, les systèmes de stockage utilisés par JOnAS au travers de son interface JDBC sont des systèmes relationnels. Les connexions JDBC sont obtenues au travers d'un connecteur JDBC. Le connecteur JDBC implémente la spécification des connecteurs J2EE, en utilisant l'interface **DataSource** ainsi qu'il est défini dans les extensions du standard JDBC 2.0. Un connecteur est configuré pour identifier une base de données et un moyen d'y accéder au travers d'un driver JDBC. Plusieurs connecteurs JDBC peuvent être déployés soit au travers du fichier `jonas.properties` ou inclus dans le répertoire autoload du service ressource. Pour des informations plus complètes sur les connecteurs dans JOnAS, se référer au Guide de programmation des connecteurs J2EE. La section suivante explique comment définir et configurer les connecteurs JDBC dans le serveur JOnAS.

Pour supporter les transactions distribuées, le connecteur JDBC nécessite l'utilisation d'au moins un driver conforme à JDBC2−XA. De tels drivers, implémentant l'interface **XADataSource** ne sont pas toujours disponibles pour les bases de données relationnelles. Le connecteur JDBC fourni avec JOnAS propose un **driver−wrapper** générique qui émule l'interface XADataSource sur un driver JDBC standard. Il est important de noter que ce driver−wrapper n'assure pas

un réel commit à deux phases pour les transactions distribuées.

Le connecteur générique de JOnAS fournit des implémentations des interfaces DriverManager, DataSource, PooledConnection, et XAConnection. Celles−ci peuvent se configurer en utilisant un driver conforme JDBC pour les serveurs de bases de données relationnelles comme Oracle, PostGres, ou MySQL.

La suite de cette section, qui décrit comment définir et configurer les connecteurs JDBC, est *specifique à JOnAS*. Cependant, la façon d'utiliser ces connecteurs JDBC dans les méthodes des composants applicatifs est standard, c. à. d. qu'elle se fait au travers des références aux ressources associées (se référer à l'exemple de la section "Ecrire des opérations d'accès BD" du guide Développer des Entity Bean).

Un fichier RAR doit être configuré et déployé (p. ex. ., Oracle1.rar pour un RAR Oracle ou MySQL1.rar pour un RAR MySQL , tels que livrés avec la plate−forme).

Pour définir une ressource "Oracle1" dans le fichier jonas.properties, ajouter le nom "Oracle1" (nom du fichier RAR) à la ligne jonas.service.resource.services ou l'inclure dans le répertoire autoload. Pour plus d'information sur le déploiement d'un RAR, se référer au Guide de programmation des connecteurs J2EE.

```
jonas.service.resource.services Oracle1,MySQL1,PostgreSQL1
```

Le fichier jonas−ra.xml qui définit une DataSource doit contenir l'information suivante :

jndiname	nom JNDI du RAR
URL	URL JDBC de la base de donnée : jdbc:<database_vendor_subprotocol>:...
dsClass	Nom de la classe implémentant le driver JDBC
user	Nom de l'utilisateur de la base de données
password	Mot de passe de l'utilisateur de la base de données

Un RAR pour Oracle configuré comme *jdbc_1* dans JNDI et utilisant le driver *JDBC 'thin'* d'Oracle, doit être décrit dans un fichier appelé `Oracle1_DM.rar`, avec les propriétés suivantes configurées dans le fichier jonas−ra.xml :

```
<jndiname>jdbc_1</jndiname>
<rarlink>JOnASJDBC_DM</rarlink>
.
.
<jonas−config−property>
  <jonas−config−property−name>user</jonas−config−property−name>
  <jonas−config−property−value>scott</jonas−config−property−value>
</jonas−config−property>
<jonas−config−property>
  <jonas−config−property−name>password</jonas−config−property−name>
  <jonas−config−property−value>tiger</jonas−config−property−value>
</jonas−config−property>
<jonas−config−property>
  <jonas−config−property−name>loginTimeout</jonas−config−property−name>
  <jonas−config−property−value></jonas−config−property−value>
</jonas−config−property>
```

```
<jonas-config-property>
  <jonas-config-property-name>URL</jonas-config-property-name>
  <jonas-config-property-value>jdbc:oracle:thin:@malte:1521:ORA1</jonas-config-property-valu
</jonas-config-property>
<jonas-config-property>
  <jonas-config-property-name>dsClass</jonas-config-property-name>
  <jonas-config-property-value>oracle.jdbc.driver.OracleDriver</jonas-config-property-value>
</jonas-config-property>
<jonas-config-property>
  <jonas-config-property-name>mapperName</jonas-config-property-name>
  <jonas-config-property-value>rdb.oracle</jonas-config-property-value>
</jonas-config-property>
```

Dans cet exemple, "malte" est le nom de système du serveur hébergeant la base de données Oracle, 1521est le numéro de port sur ce serveur, et ORA1 est le ORACLE_SID.

Cet exemple utilise le driver JDBC 'thin' d'Oracle. Pour un serveur d'application tournant sur le même système que la base Oracle, il est possible d'utiliser le driver JDBC OCI d'Oracle; dans ce cas, l'URL à utiliser est jdbc:oracle:oci7: ou jdbc:oracle:oci8:, suivant la version d'Oracle. Les drivers JDBC d'Oracle peuvent être téléchargés depuis leur site Web.

Pour créer un RAR MySQL configuré comme *jdbc_2* dans JNDI, il doit être décrit dans un fichier nommé `MySQL2_DM.rar`, avec les propriétés suivantes configurées dans le fichier jonas−ra.xml :

```
<jndiname>jdbc_2</jndiname>
<rarlink>JOnASJDBC_DM</rarlink>
.
.
<jonas-config-property>
  <jonas-config-property-name>user</jonas-config-property-name>
  <jonas-config-property-value></jonas-config-property-value>
</jonas-config-property>
<jonas-config-property>
  <jonas-config-property-name>password</jonas-config-property-name>
  <jonas-config-property-value></jonas-config-property-value>
</jonas-config-property>
<jonas-config-property>
  <jonas-config-property-name>loginTimeout</jonas-config-property-name>
  <jonas-config-property-value></jonas-config-property-value>
</jonas-config-property>
<jonas-config-property>
  <jonas-config-property-name>URL</jonas-config-property-name>
  <jonas-config-property-value>jdbc:mysql://malte/db_jonas</jonas-config-property-value>
</jonas-config-property>
<jonas-config-property>
  <jonas-config-property-name>dsClass</jonas-config-property-name>
  <jonas-config-property-value>org.gjt.mm.mysql.Driver</jonas-config-property-value>
</jonas-config-property>
<jonas-config-property>
  <jonas-config-property-name>mapperName</jonas-config-property-name>
  <jonas-config-property-value>rdb.mysql</jonas-config-property-value>
```

```
</jonas-config-property>
```

Pour créer un RAR PostGreSQL configuré comme *jdbc_3* dans JNDI, il doit être décrit dans un fichier nommé `PostGreSQL3_DM.rar`, avec les propriétés suivantes configurées dans le fichier jonas-ra.xml :

```
<jndiname>jdbc_3</jndiname>
<rarlink>JOnASJDBC_DM</rarlink>
.
.
<jonas-config-property>
  <jonas-config-property-name>user</jonas-config-property-name>
  <jonas-config-property-value>jonas</jonas-config-property-value>
</jonas-config-property>
<jonas-config-property>
  <jonas-config-property-name>password</jonas-config-property-name>
  <jonas-config-property-value>jonas</jonas-config-property-value>
</jonas-config-property>
<jonas-config-property>
  <jonas-config-property-name>loginTimeout</jonas-config-property-name>
  <jonas-config-property-value></jonas-config-property-value>
</jonas-config-property>
<jonas-config-property>
  <jonas-config-property-name>URL</jonas-config-property-name>
  <jonas-config-property-value>jdbc:postgresql:/malte:5432/db_jonas</jonas-config-property-v
</jonas-config-property>
<jonas-config-property>
  <jonas-config-property-name>dsClass</jonas-config-property-name>
  <jonas-config-property-value>org.postgresql.Driver</jonas-config-property-value>
</jonas-config-property>
<jonas-config-property>
  <jonas-config-property-name>mapperName</jonas-config-property-name>
  <jonas-config-property-value>rdb.mpostgres</jonas-config-property-value>
</jonas-config-property>
```

Le nom et le mot de passe de l'utilisateur base de données peuvent être gérés de deux façons : 1) mis dans le fichier jonas-ra.xml dans l'archive RAR, les composants applicatifs utilisent alors la méthode getConnection() , ou 2) non positionnés dans l'archive RAR et laisser les composants applicatifs utiliser la méthode getConnection(String username, String password) .

CMP2.0/JORM

Pour implémenter la persistence EJB 2.0 (CMP2.0), JOnAS repose sur le framework JORM. JORM doit adapter sa correspondance Objet/Relationnel à la base sur laquelle il s'appuie et utilise pour cela des adapteurs appelés "mappers". Pour chaque type de base de donnée (et plus précisément pour chaque driver JDBC), le mapper correspondant doit être spécifié dans le fichier jonas-ra.xml de l'archive RAR déployée. L'élément ***mapperName*** est fourni dans ce but.

property name	description	possible values

mapperName	JORM database mapper	• rdb: mapper générique (driver standard JDBC ...) • rdb.postgres: mapper pour PostgreSQL • rdb.oracle8: mapper pour Oracle 8 et versions antérieures • rdb.oracle: mapper pour Oracle 9 • rdb.mckoi: mapper pour McKoi Db • rdb.mysql: mapper pour MySQL • rdb.sqlserver: mapper pour un Server MS SQL • Se référer à la documentation JORM pour une liste complète et à jour.

Configuration du ConnectionManager

Chaque RAR utilise un gestionnaire de connexions qui peut être configuré à l'aide des propriétés additionnelles décrites dans la table ci−dessous. Le fichier Postgres1.jonas−ra.xml fournit un exemple des paramètres possibles. Ces paramètres sont optionnels et ont des valeurs par défaut.

pool−params elements

property name	description	default value
pool−init	Nombre initial de connexions	0
pool−min	Nombre Minimal de connexions	0
pool−max	Nombre Maximal de connexions	−1 (unlimited)
pool−max−age	Nombre de millisecondes pour garder la connexion	0 (unlimited)
pstmt−max	Nombre maximum de PreparedStatements mises en cache par connexion	10

jdbc−conn−params elements

property name	description	default value
jdbc−check−level	Niveau de vérification JDBC	0 (pas de vérification)
jdbc−test−statement	Requête de test	

jdbc−test−statement n'est pas utilisé quand jdbc−check−level est égal à 0 ou 1.

Tracer les requêtes SQL avec P6Spy

L'outil P6Spy est intégré à JOnAS pour fournir un environnement de trace des requêtes SQL envoyées à la base de donnée.

Pour valider cette fonctionnalité de trace, respecter les étapes suivantes :

- Mettre à jour le fichier approprié jonas−ra.xml de l'archive RAR en positionnant la propriété `dsClass` à `com.p6spy.engine.spy.P6SpyDriver`.

- Positionner la propriété `realdriver` du fichier `spy.properties` (localisée dans $JONAS_BASE/conf) au driver jdbc de votre base de donnée.
- Vérifier que `logger.org.objectweb.jonas.jdbc.sql.level` est positionné à DEBUG dans $JONAS_BASE/conf/trace.properties.

Exemple:

Contenu du fichier :

```
<jonas-resource>
  <jndiname>jdbc_3</jndiname>
  <rarlink>JOnASJDBC_DM</rarlink>
  <native-lib></native-lib>
  <log-enabled>true</log-enabled>
  <log-topic>org.objectweb.jonas.jdbc.DMPostgres</log-topic>
  <pool-params>
    <pool-init>0</pool-init>
    <pool-min>0</pool-min>
    <pool-max>100</pool-max>
    <pool-max-age>0</pool-max-age>
    <pstmt-max>10</pstmt-max>
  </pool-params>
  <jdbc-conn-params>
    <jdbc-check-level>0</jdbc-check-level>
    <jdbc-test-statement></jdbc-test-statement>
  </jdbc-conn-params>
  <jonas-config-property>
    <jonas-config-property-name>user</jonas-config-property-name>
    <jonas-config-property-value>jonas</jonas-config-property-value>
  </jonas-config-property>
  <jonas-config-property>
    <jonas-config-property-name>password</jonas-config-property-name>
    <jonas-config-property-value>jonas</jonas-config-property-value>
  </jonas-config-property>
  <jonas-config-property>
    <jonas-config-property-name>loginTimeout</jonas-config-property-name>
    <jonas-config-property-value></jonas-config-property-value>
  </jonas-config-property>
  <jonas-config-property>
    <jonas-config-property-name>URL</jonas-config-property-name>
    <jonas-config-property-value>jdbc:postgresql://your_host:port/your_db</jonas-config-prope
  </jonas-config-property>
  <jonas-config-property>
    <jonas-config-property-name>dsClass</jonas-config-property-name>
    <jonas-config-property-value>com.p6spy.engine.spy.P6SpyDriver</jonas-config-property-valu
  </jonas-config-property>
  <jonas-config-property>
    <jonas-config-property-name>mapperName</jonas-config-property-name>
    <jonas-config-property-value>rdb.postgres</jonas-config-property-value>
  </jonas-config-property>
  <jonas-config-property>
    <jonas-config-property-name>logTopic</jonas-config-property-name>
```

```
      <jonas-config-property-value>org.objectweb.jonas.jdbc.DMPostgres</jonas-config-property-va
    </jonas-config-property>
  </jonas-resource>
```

Dans le fichier $JONAS_BASE/conf/spy.properties :

```
realdriver=org.postgresql.Driver
```

Dans $JONAS_BASE/conf/trace.properties:

```
logger.org.objectweb.jonas.jdbc.sql.level  DEBUG
```

Migration du Service DBM vers le JDBC RA

La migration du fichier Database.properties vers un connecteur similaire peut se faire en utilisant l'outil RAConfig. Se référer à la description de RAConfig pour plus de précision.

```
RAConfig -dm -p MySQL $JONAS_ROOT/rars/autoload/JOnAS_jdbcDM MySQL
```

Cette commande va créer un fichier MySQL.rar basé sur le fichier MySQL.properties, ainsi qu'il est spécifié par le paramètre -p. Il incluera également le <rarlink> vers le JOnAS_jdbcDM.rar, ainsi que le spécifie le paramètre -dm.

Le jonas-ra.xml créé par la commande précédente pourra être mis à jour ultérieurement, si nécessaire. Une fois les propriétés additionnelles configurées, mettre à jour le fichier MySQL.rar en utilisant la ligne de commande suivante :

```
RAConfig -u jonas-ra.xml MySQL
```

Guide du Programmeur d'Application J2EE

Public auquel il est destiné et son contenu

Le public auquel ce guide est destiné est le fournisseur de composant applicatif, i.e. la personne en charge du développement des composants applicatifs côté serveur (le niveau métier).

Le contenu de ce guide est le suivant :

Principes

JOnAS supporte deux types de composants applicatifs J2EE : les **Enterprise Beans** et les **composants Web**. En plus de fournir un guide de construction de composants applicatifs, ce guide concerne également leur assemblage, leur déploiement ainsi que leur administration.

Création d'Entreprise Bean

La personne en charge du développement d' Enterprise Beans peut consulter le Guide du Programmeur d'Entreprise Bean pour prendre connaissance de la façon de réaliser les tâches suivantes :

1. Ecrire le code source pour les beans.
2. Spécifier le descripteur de déploiement.
3. Assembler les classes compilées et le descripteur de déploiement en un fichier JAR EJB.

La documentation JOnAS fournit des guides pour développer les trois types d' enterprise bean :

- Session beans
- Entity beans
- Message driven beans

La spécification du descripteur de déploiement est presentée au chapitre Définir le Descripteur de Déploiement.

Des éléments plus spécifiques relatifs au comportement transactionnel, à l'environnement des Enterprise Bean , ou au service de sécurité, sont presentés dans les chapitres correspondants : Comportement des Transactions, Environment des Entreprise Bean, Gestion de la Sécurité.

Les principes et outils destinés à la création des fichiers JAR EJB sont presentés aux chapitres Packaging d'EJB et Guide de Déploiement et d'Administration.

Création de Composants Web

Les concepteurs Web en charge des développements de pages JSP et de logiciels fournissant des servlets peuvent consulter le Guide du Programmeur d'Application Web.

Le guide Développement de Composants Web explique aussi bien comment construire des composants Web, que comment accéder aux Enterprise Beans depuis les composants Web.

La spécifiation du descripteur de Déploiement est présentée au chapitre Définir le Descripteur de Déploiement Web.

Les composants Web peuvent être utilisés en tant que composants d' *application Web* ou en tant que composants d' *application J2EE*. Dans les deux cas, un fichier WAR sera créé , mais son contenu sera différent dans ces deux situations. Dans le premier cas, le fichier WAR contient les composants Web et les Enterprise Beans. Dans le second cas, le fichier WAR ne contient pas les Enterprise Beans. Le fichier JAR EJB contenant les Enterprise Beans est emballé avec le fichier WAR contenant les composants Web, en un fichier EAR.
Les principes et outils servant à créer des fichiers WAR sont présentés dans les guides suivants Création d'un Package WAR et Guide de Déploiement et d'Installation.

Intégrateur d'Application J2EE

L'intégrateur d'application en charge d'assembler les composants applicatifs déjà regroupés depuis les fichiers JAR EJB et WAR en un fichier EAR J2EE, pourra obtenir des informations utiles au chapitre Guide de l'Intégrateur d'Application J2EE.

Déployeur d'Application et Administrateur

JOnAS fournit des outils pour le déploiement et l'administration d'Enterprise Beans (EJB JARs), d'applications Web (WARs), et d'applications J2EE (EARs).

Le Guide du Déploiement et d'Installation recouvre les questions relatives au déploiement de composants applicatifs.

Le Guide d'Administration présente les informations relatives au mode d'administration du serveur JOnAS et des services JOnAS qui permettent le déploiement de différents types de composants applicatifs : le service de conteneur EJB, le service de conteneur Web, et le service EAR.

Hiérarchie de chargeurs de classe JOnAS

Cette partie décrit une caractéristique nouvelle et importante de l'intégration J2EE : la hiérarchie de chargeurs de classe JOnAS.

Compréhension de la hiérarchie de chargeur de classe

Une application est déployée par son propre chargeur de classe. Cela signifie, par exemple, que si un fichier WAR et un fichier JAR EJB sont déployés séparément, les classes contenues dans les deux archives sont chargées avec deux chargeurs de classe sans lien de hiérarchie entre eux. Ainsi, les EJB contenus dans le fichier JAR ne pourront être vus des composants Web contenus dans le WAR. Ceci n'est pas acceptable, en particulier dans le cas où les composants Web d'une application auraient besoin de faire référence et d'utiliser certains des EJBs (cela concerne évidemment des références locales dans une meme JVM).

Pour cette raison, avant l'utilisation des fichiers EAR, si l'on devait déployer une application Web qui utilise les EJBs, le fichier JAR EJB devait être situé dans la bibliothèque *WEB−INF/lib* de l' application Web.

Actuellement, avec l'intégration J2EE et l'utilisation du packaging du type EAR, les problèmes de visibilité de classes n'existent plus et la présence du fichier EJB JAR dans la bibliothèque *WEB−INF/lib* n'est plus nécessaire.

La partie suivante décrit la hiérarchie de chargeurs de classe JOnAS et explique le mécanisme utilisé pour retrouver les classes référencées.

Chargeur de classe initialiseur

Le chargeur de classe initialiseur (bootstrap) est un chargeur de classe spécifique à JOnAS qui va charger toutes les classes dont a besoin JOnAS pour démarrer. Ce chargeur de classe a le chargeur de classe système pour chargeur de classe parent. Le chargeur de classe initialiseur ajoute toutes les bibliothèques nécessaires au lancement du serveur JOnAS (bibliothèques pour la messagerie, tomcat, etc.); il charge également les classes situées dans `XTRA_CLASSPATH`. Par exemple, si on charge un composant EJB non packagé, l'emplacement de ses classes devra être spécifié dans `XTRA_CLASSPATH`.

Pour avoir une bibliothèque disponible pour chaque composant s'exécutant à l'intérieur de JOnAS, ajouter les fichiers jars nécessaires dans la bibliothèque `JONAS_ROOT/lib/ext` ou dans `JONAS_BASE/lib/ext`. Les jars contenus dans `JONAS_BASE/lib/ext` sont chargés en premier, suivis par ceux contenus dans `JONAS_ROOT/lib/ext`. Tous les jars situés dans les sous−bibliothèques seront également chargés.
Si un jar particulier n'est nécessaire qu'à une application web (i.e., nécessite d'utiliser une version de fichier jar différente de celle chargée par JOnAS), changer la conformité du chargement de classe de l'application web suivant le modèle de délégation java 2. Voir ci−dessous: chargement de classe WEB.
Il est également possible d'utiliser le mécanisme d'extension, qui est décrit dans la section dépendances de la spécification J2EE (paragraphe 8.1.1.28)

Chargeurs de classe JOnAS

La hiérarchie des chargeurs de classe JOnAS qui permet le déploiement d'applications EAR sans placer les fichiers JAR EJB dans la bibliothèque *WEB−INF/lib* consiste en :

chargeur de classe EAR

Le chargeur de classe EAR est responsable du chargement d'application EAR. Il n'y a qu'un chargeur de classe EAR par application EAR. Ce chargeur de classe est l'enfant du chargeur de classe initialiseur, lui rendant ainsi visible les classes JOnAS.

chargeur de classe EJB

Le chargeur de classe EJB est responsable du chargement de tous les JARs EJB de l'application EAR, ainsi tous les EJBs d'une même application EAR sont chargés avec le même chargeur de classe EJB. Ce chargeur de classe est l'enfant du chargeur de classe EAR.

chargeur de classe WEB

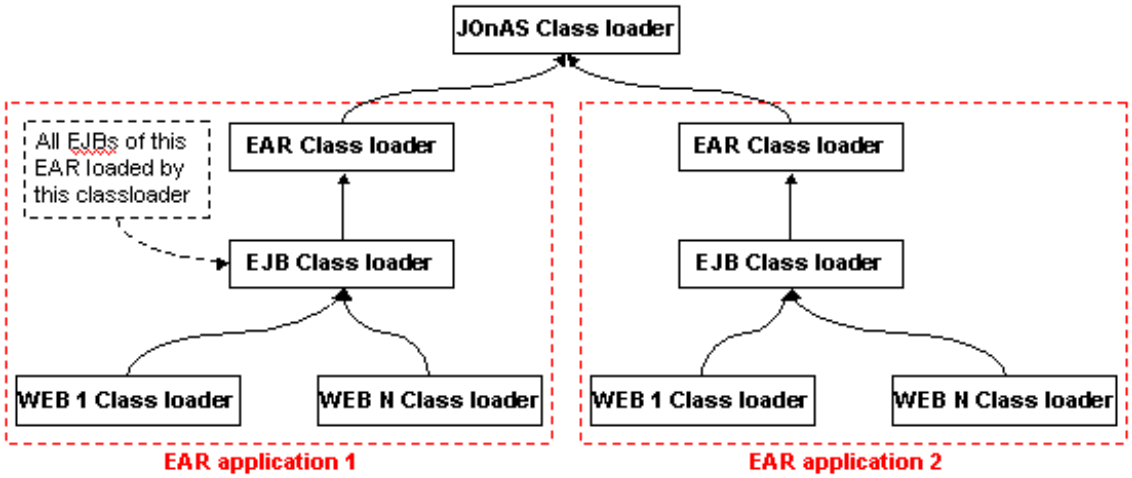
Le chargeur de classe WEB est responsable du chargement des composants Web. Il y a un chargeur de classe WEB par fichier WAR, et ce chargeur de classe est l'enfant du chargeur de classe EJB. Le fait d'utiliser cette hiérarchie de chargeurs de classe (le chargeur de classe EJB est le parent du chargeur de classe WEB) permet d'éliminer le problème de visibilité entre les classes quand un composant WEB essaie de faire référence à des EJBs; les classes chargées avec le chargeur de classe EJB sont rendues visibles des classes chargées par leurs chargeurs de classe fils (chargeur de classe WEB).

La conformité du chargeur de classe de l'application web avec le modèle de délégation java 2 peut être modifiée en utilisant le ficher jonas−web.xml. Ceci est décrit au chapitre "Personnaliser le fichier jonas−web.xml."

Si l'élément java2−delegation−model est positionné à faux, le chargeur de classe de l'application web cherche la classe dans son propre référentiel avant d'interroger son chargeur de classe parent.

Conclusion

Le modèle résultant de hiérarchie de chargeurs de classe est le suivant :


JOnAS Class loader
All EJBs of this EAR loaded by this classloader
EAR Class loader
EJB Class loader
WEB 1 Class loader
WEB N Class loader
EAR application 1
EAR Class loader
EJB Class loader
WEB 1 Class loader
WEB N Class loader
EAR application 2

Guide du programmeur EJB: Développer des Beans Sessions

Public visé et contenu

Ce guide est destiné aux fournisseurs de composants EJB, c'est−à−dire aux personnes chargées de développer des composants logiciels côté serveur et plus particulièrement des Beans Sessions.

Contenu de ce guide:

Introduction

Pour développer un Bean Session, le fournisseur de composants EJB doit fournir les éléments suivants :

- **L'interface du composant (Component Iterface)** est ce que le client voit du composant. Il contient l'ensemble des "méthodes métiers" de l'EJB.
- **L'interface d'accueil (Home Interface)** contient les méthodes permettant au client de gérer le cycle de vie (création, ...) de l'EJB.
- **La classe d'implémentation de l'EJB** implémente les méthodes métiers ainsi que l'ensemble des méthodes (comme défini dans la spécification EJB) permettant au conteneur de gérer ces composants.
- **Le descripteur de déploiement** contient les propriétés du composant qui peuvent être configurées au moment de l'intégration ou du déploiement.

Notez que selon la spécification EJB 2.0, le couple "interface du composant et interface d'accueil" peut être local ou distant. **Les interfaces locales** (Composant et Accueil) sont utilisées par les clients s'exécutant dans la même JVM que le composant EJB. Les méthodes de création (create) ou de recherche (finders) de l'interface d'accueil retournent alors respectivement des interfaces locales ou distantes. Un composant EJB peut avoir indépendamment les deux types d'interfaces, distantes ou locales, même si en général un seul type est utilisé.

La description de ces éléments est fournie dans la partie suivante.

Note: Dans cette documentation les termes Beans et Composants désignent toujours des composants EJB.

Un Bean Session est un composant de courte vie fournissant des services aux clients. Il existe des Beans Sessions **sans état (Stateless)** et **avec état (Stateful)**. Les Beans Sessions Stateless ne maintiennent aucun état conversationnel entre deux appels successifs. Chaque instance de Bean Session Stateless peut être utilisée n'importe quand, par n'importe

quel client. Les Beans Sessions Stateful maintiennent un état conversationnel entre les appels. Chaque Bean Session Stateful est associé à un seul client. Un Bean Session Stateful dont les transactions sont gérées par le conteneur peut optionnellement implémenter l'interface **SessionSynchronization**. Dans ce cas, le Bean sera informé de l'issue de la transaction. Un "rollback" d'une transaction peut rendre l'état d'un Bean Session incohérent; dans ce cas, implémenter l'interface SessionSynchronization peut permettre au composant de mettre son état à jour selon l'issue de la transaction.

L'interface d'accueil

L'interface d'accueil (Home interface) d'un Bean Session, définie une ou plusieurs méthodes *create(...)*. Chaque méthode *create* doit être nommée `create` et doit correspondre à une méthode *ejbCreate* définie dans la classe d'implémentation de l'EJB. Le type de retour de ces méthodes *create* doit être une interface du composant EJB. L'interface d'accueil d'un Bean Session Stateless doit avoir une méthode *create* qui ne prend pas d'arguments.

Toutes les Exceptions susceptibles d'être déclenchées par la méthode *ejbCreate* (clause "throws") doivent également être définies dans la clause "throws" de la méthode *create* correspondante de l'interface d'accueil.

L'interface d'accueil distante hérite de l'interface `javax.ejb.EJBHome` , alors que **l'interface d'accueil locale** hérite de `javax.ejb.EJBLocalHome`.

Exemple:

L'exemple suivant montre l'interface d'accueil d'un EJB appelé `Op`.

```
public interface OpHome extends EJBHome {
    Op create(String user) throws CreateException, RemoteException;
}
```

L'interface d'accueil locale peut également être définie (L'interface locale du composant étant appelée `LocalOp`)

```
public interface LocalOpHome extends EJBLocalHome {
    LocalOp create(String user) throws CreateException;
}
```

L'interface du composant

L'interface du composant est la vue client d'une instance d'un Bean Session. Cette interface contient l'ensemble des méthodes métier d'un EJB. Cette interface doit hériter de l'interface `javax.ejb.EJBObject` si elle est distante, ou de l'interface `javax.ejb.EJBLocalObject` si elle est locale. Les méthodes définies dans l'interface distante doivent suivre les régles de Java RMI (c'est−à−dire que leurs arguments et leurs types de retour doivent être des types valides pour RMI, et que leur clause "throws" doit inclure `java.rmi.RemoteException`). Pour chaque méthode définie dans l'interface du composant, il doit y avoir la méthode correspondante dans la classe d'implémentation de l'EJB (même nom, même signature, même liste d'exceptions, mis à part pour RemoteException).

Exemple:

```
public interface Op extends EJBObject {
    public void buy (int Shares)  throws RemoteException;
    public int  read ()           throws RemoteException;
}
```

Cette interface peut également être déclarée locale :

```
public interface LocalOp extends EJBLocalObject {
    public void buy (int Shares);
    public int  read ();
}
```

La classe d'implémentation de l'EJB

Cette classe implémente les méthodes métier définies dans l'interface du composant ainsi que les méthodes de l'interface SessionBean, qui sont dédiées au conteneur d'EJB. La classe doit être déclarée publique et ne peut pas être abstraite. L'interface *SessionBean* définie les méthodes suivantes:

- *public void **setSessionContext**(SessionContext ic);*

  Cette méthode est utilisée par le conteneur pour passer à l'instance du Bean une référence au SessionContext. Le conteneur invoque cette méthode sur chaque instance qu'il a nouvellement créée. Généralement cette méthode stocke cette référence dans une variable d'instance.
- *public void **ejbRemove**();*

  Cette méthode est invoquée par le conteneur quand une instance va être détruite. Comme la plupart des Beans Sessions n'ont pas de ressources externes à relacher, cette méthode est souvent laissée vide.
- *public void **ejbPassivate**();*

  Cette méthode est invoquée par le conteneur lorsqu'il souhaite passiver l'instance. Après l'exécution de cette méthode, l'instance doit être dans un état qui permette au conteneur de sauvegarder son état en utilisant le protocole se sérialisation d'objet Java.
- *public void **ejbActivate**();*

  Cette méthode est invoquée par le conteneur quand une instance vient juste d'être réactivée. L'instance peut alors récupérer toutes les ressources qu'elle avait relâchées lors de l'appel de la méthode ejbPassivate

Un Bean Session Stateful dont les transactions sont gérées par le conteneur peut optionnellement implémenter l'interface `javax.ejb.SessionSynchronization`. Cette interface peut fournir au Bean des notifications à propos du déroulement de la transaction. Les méthodes de l'interface *SessionSynchronization* sont les suivantes :

- *public void **afterBegin**();*

Cette méthode notifie une instance d'un Bean Session qu'une nouvelle transaction vient de démarrer.

- *public void **afterCompletion(**boolean committed);*

  Cette méthode notifie un Bean Session que la phase de commit de la transaction est terminée et dit à l'instance si la transaction a validé correctement ou a effectué un rollback.
- *public void **beforeCompletion**();*

  Cette méthode notifie un Bean Session qu'une transaction va être validée.

Exemple:

```
package sb;

import java.rmi.RemoteException;
import javax.ejb.EJBException;
import javax.ejb.EJBObject;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.ejb.SessionSynchronization;
import javax.naming.InitialContext;
import javax.naming.NamingException;

// Un exemple d'un Bean Session Statefull, implémentant SessionSynchronization.

public class OpBean implements SessionBean, SessionSynchronization {

    protected int total = 0;        // état actuel du Bean
    protected int newtotal = 0;        // valeur à l'intérieur de la transaction, non commitée
    protected String clientUser = null;
    protected SessionContext sessionContext = null;

    public void  ejbCreate(String user) {
        total = 0;
        newtotal = total;
        clientUser = user;
    }

    public void ejbActivate() {
        // rien à faire ici pour cet exemple
    }

    public void ejbPassivate() {
        // rien à faire ici pour cet exemple
    }

    public void ejbRemove() {
        // rien à faire ici pour cet exemple
    }

    public void setSessionContext(SessionContext sessionContext) {
        this.sessionContext = sessionContext;
    }
```

```
    public void afterBegin() {
        newtotal = total;
    }

    public void beforeCompletion() {

        // On peut accéder à l'environnement du bean de partout,
        // par exemple d'ici!
        try {
            InitialContext ictx = new InitialContext();
            String value = (String) ictx.lookup("java:comp/env/prop1");
            // cette valeur doit être définie dans le ejb-jar.xml
        } catch (NamingException e) {
            throw new EJBException(e);
        }
    }

    public void afterCompletion(boolean committed) {
        if (committed) {
            total = newtotal;
        } else {
            newtotal = total;
        }
    }

    public void buy(int s) {
        newtotal = newtotal + s;
        return;
    }

    public int read() {
        return newtotal;
    }
}
```

Ajuster le pool de Bean Session Stateless

JOnAS gére un pool d'instances pour chaque Bean Session Stateless. Ce pool peut être configuré dans le descripteur de déploiement spécifique de JOnAS grâce aux tags suivant :

min−pool−size

Cet entier optionnel représente le nombre minimum d'instances que va créer le pool lors du chargement de l'EJB. Cela va augmenter le temps de création du Bean, au moins pour le premier. La valeur par défaut est 0.

max−cache−size

Cet entier optionnel représente le nombre maximum d'instances en mémoire. L'objectif de cette valeur est de garder JOnAS scalable. La politique est la suivante :
Au moment de la création d'un Bean, une instance est prise du pool d'instances disponibles. Si le pool est vide, une instance est toujours créée. Quand l'instance doit être relâchée (à la fin de l'exécution de la méthode métier), elle est remise dans le pool, à moins que le nombre d'instances créées excède la valeur de `max−cache−size`, dans ce cas l'instance est supprimée. La valeur par défaut est "pas de limite".

exemple

```
<jonas−ejb−jar>
  <jonas−session>
    <ejb−name>SessSLR</ejb−name>
    <jndi−name>EJB/SessHome</jndi−name>
    <max−cache−size>20</max−cache−size>
    <min−pool−size>10</min−pool−size>
  </jonas−session>
</jonas−ejb−jar>
```

EJB Programmer's Guide: Developing Entity Beans

Target Audience and Content

The target audience for this guide is the Enterprise Bean provider, i.e. the person in charge of developing the software components on the server side, and more specifically the Entity Beans.

The content of this guide is the following:

Introduction

An Entity Bean is comprised of the following elements, which are developed by the Enterprise Bean Provider:

- The **Component Interface** is the client view of the bean. It contains all the "business methods" of the bean.
- The **Home Interface** contains all the methods for the bean life cycle (creation, suppression) and for instance retrieval (finding one or several bean objects) used by the client application. It can also contain methods called "home methods," supplied by the bean provider, for business logic which is not specific to a bean instance.
- The **Primary Key class** (for entity beans only) contains a subset of the bean's fields that identifies a particular instance of an entity bean. This class is optional since the bean programmer can alternatively choose a standard class (for example, java.lang.String)
- The **bean implementation class** implements the business methods and all the methods (described in the EJB specification) allowing the bean to be managed in the container.
- The **deployment descriptor**, containing the bean properties that can be edited at assembly or deployment

time.

Note that, according to the EJB 2.0 specification, the couple "Component Interface and Home Interface" can be either local or remote. **Local Interfaces** (Home and Component) are to be used by a client running in the same JVM as the EJB component. Create and finder methods of a local (or remote) home interface return local (or remote) component interfaces. An EJB component may have both remote and local interfaces, even if normally only one type of interface is provided. If an entity bean is the target of a container−managed relationship (refer to EJB 2.0 persistence), then it must have local interfaces.

The description of these elements is provided in the following sections.

Note that in this documentation, the term "Bean" always means "Enterprise Bean."

An entity bean represents persistent data. It is an object view of an entity stored in a relational database. The persistence of an entity bean can be handled in two ways:

- **Container−Managed Persistence**: the persistence is implicitly managed by the container, no code for data access is supplied by the bean provider. The bean's state will be stored in a relational database according to a mapping description delivered within the deployment descriptor (CMP 1.1) or according to an implicit mapping (CMP 2.0).
- **Bean−Managed Persistence**: the bean provider writes the database access operations (JDBC code) in the methods of the enterprise bean that are specified for data creation, load, store, retrieval, and remove operations (ejbCreate, ejbLoad, ejbStore, ejbFind..., ejbRemove).

Currently, the platform handles persistence in relational storage systems through the JDBC interface. For both container−managed and bean−managed persistence, JDBC connections are obtained from an object provided at the EJB server level, the **DataSource**. The DataSource interface is defined in the JDBC 2.0 standard extensions. A DataSource object identifies a database and a means to access it via JDBC (a JDBC driver). An EJB server may propose access to several databases and thus provides the corresponding DataSource objects. DataSources are described in more detail in the section "Configuring JDBC DataSources."

The Home Interface

In addition to "home business methods," the Home interface is used by any client application to create, remove, and retrieve instances of the entity bean. The bean provider only needs to provide the desired interface; the container will automatically provide the implementation. The interface must extend the `javax.ejb.EJBHome` interface if it is remote, or the `javax.ejb.EJBLocalHome` interface if it is local. The methods of a remote home interface must follow the rules for java RMI. The signatures of the "create" and "find..." methods should match the signatures of the "ejbCreate" and "ejbFind..." methods that will be provided later in the enterprise bean implementation class (same number and types of arguments, but different return types).

create methods:

- The return type is the enterprise bean's component interface.

- The exceptions defined in the throws clause must include the exceptions defined for the ejbCreate and ejbPostCreate methods, and must include `javax.ejb.CreateException` and `java.rmi.RemoteException` (the latter, only for a remote interface).

remove methods:

- The interfaces for these methods must not be defined, they are inherited from EJBHome or EJBLocalHome.
- The method is `void remove` taking as argument the primary key object or the handle (for a remote interface).
- The exceptions defined in the throws clause should be `javax.ejb.RemoveException` and `java.rmi.RemoteException` for a remote interface.
- The exceptions defined in the throws clause should be `javax.ejb.RemoveException` and `java.ejb.EJBException` for a local interface.

finder methods:

Finder methods are used to search for an EJB object or a collection of EJB objects. The arguments of the method are used by the entity bean implementation to locate the requested entity objects. For bean−managed persistence, the bean provider is responsible for developing the corresponding ejbFinder methods in the bean implementation. For container−managed persistence, the bean provider does not write these methods; they are generated at deployment time by the platform tools; the description of the method is provided in the deployment descriptor, as defined in the section "Configuring database access for container−managed persistence." In the Home interface, the finder methods must adhere to the following rules:

- They must be named "*find<method>*" (e.g. `findLargeAccounts`).
- The return type must be the enterprise bean's component interface, or a collection thereof.
- The exceptions defined in the throws clause must include the exceptions defined for the matching ejbFind method, and must include `javax.ejb.FinderException` and `java.rmi.RemoteException` (the latter, only for a remote interface).

At least one of these methods is mandatory: *findByPrimaryKey*, which takes as argument a primary key value and returns the corresponding EJB object.

home methods:

- Home methods are methods that the bean provider supplies for business logic that is not specific to an entity bean instance.
- The throws clause of every home method on the remote home interface includes the java.rmi.RemoteException.
- Home methods implementation is provided by the bean developer in the bean implementation class as public static methods named `ejbHome<METHOD_NAME>(...)`, where <METHOD_NAME> is the name of the method in the home interface.

Example

The Account bean example, provided with the platform examples, is used to illustrate these concepts. The state of an entity bean instance is stored in a relational database, where the following table should exist, if CMP 1.1 is used:

```
create table ACCOUNT (ACCNO integer primary key, CUSTOMER varchar(30), BALANCE
number(15,4));

public interface AccountHome extends EJBHome {

    public Account create(int accno, String customer, double balance)
        throws RemoteException, CreateException;

    public Account findByPrimaryKey(Integer pk)
        throws RemoteException, FinderException;

    public Account findByNumber(int accno)
        throws RemoteException, FinderException;

    public Enumeration findLargeAccounts(double val)
        throws RemoteException, FinderException;
}
```

The Component Interface

Business methods:

The Component Interface is the client's view of an instance of the entity bean. It is what is returned to the client by the Home interface after creating or finding an entity bean instance. This interface contains the business methods of the enterprise bean. The interface must extend the `javax.ejb.EJBObject` interface if it is remote, or the `javax.ejb.EJBLocalObject` if it is local. The methods of a remote component interface must follow the rules for java RMI. For each method defined in this component interface, there must be a matching method of the bean implementation class (same arguments number and types, same return type, same exceptions except for RemoteException).

Example

```
public interface Account extends EJBObject {
    public double getBalance() throws RemoteException;
    public void setBalance(double d) throws RemoteException;
    public String getCustomer() throws RemoteException;
    public void setCustomer(String c) throws RemoteException;
    public int getNumber() throws RemoteException;
}
```

The Primary Key Class

The Primary Key class is necessary for entity beans only. It encapsulates the fields representing the primary key of an entity bean in a single object. If the primary key in the database table is composed of a single column with a basic data type, the simplest way to define the primary key in the bean is to use a standard java class (for example, `java.lang.Integer` or `java.lang.String`). This must have the same type as a field in the bean class. It is not possible to define it as a primitive field (for example, int, float or boolean). Then, it is only necessary to specify the type of the primary key in the deployment descriptor:

```
<prim-key-class>java.lang.Integer</prim-key-class>
```

And, for container-managed persistence, the field which represents the primary key:

```
<primkey-field>accno</primkey-field>
```

The alternative way is to define its own Primary Key class, described as follows:

The class must be serializable and must provide suitable implementation of the `hashcode()` and `equals(Object)` methods.

For *container-managed persistence*, the following rules must be followed:

- The fields of the primary key class must be declared as public.
- The primary key class must have a public default constructor.
- The names of the fields in the primary key class must be a subset of the names of the container-managed fields of the enterprise bean.

Example

```
public class AccountBeanPK implements java.io.Serializable {

      public int accno;

      public AccountBeanPK(int accno) { this.accno = accno; }

      public AccountBeanPK() { }

      public int hashcode() { return accno; }

      public boolean equals(Object other) {

            ...

      }
```

```
}
```

Special case: Automatic generation of Integer primary keys field

For a CMP 2.0 and CMP1 entity bean, the JOnAS−specific deployment descriptor contains an additional, optional element, `automatic−pk,` which may have one of the following values: `true or false (default value).`
If the value is `true`, the primary key field's value of the bean is automatically generated by the container when the entity bean is created. The primary type java.lang.Integer must be selected for the primary key's field in the <prim−key−class> tag, and the value of this field in the ejbCreate method should not be initialized.

Example of using the new element:

JOnAS specific deployment descriptor:

```
 <jonas−entity>
    <ejb−name>AddressEJB</ejb−name>
    <jdbc−mapping>
        <jndi−name>jdbc_1</jndi−name>
        <automatic−pk>true</automatic−pk>
    </jdbc−mapping>
 </jonas−entity>
```

Standard deployment descriptor:

```
  <entity>
      <ejb−name>AddressEJB</ejb−name>
      <local−home>com.titan.address.AddressHomeLocal</local−home>
      <local>com.titan.address.AddressLocal</local>
      <ejb−class>com.titan.address.AddressBean</ejb−class>
      <persistence−type>Container</persistence−type>
      <prim−key−class>java.lang.Integer</prim−key−class>
      <reentrant>False</reentrant>
      <cmp−version>2.x</cmp−version>
      <abstract−schema−name>Cmp2_Address</abstract−schema−name>
      <cmp−field><field−name>id</field−name></cmp−field>
      <cmp−field><field−name>street</field−name></cmp−field>
      <cmp−field><field−name>city</field−name></cmp−field>
      <cmp−field><field−name>state</field−name></cmp−field>
      <cmp−field><field−name>zip</field−name></cmp−field>
      <primkey−field>id</primkey−field>
```

Address Bean Class:

```
   // Field Id is not initialized during ejbCreate methode
   public Integer ejbCreateAddress(String street, String city, String state,
String zip ) throws javax.ejb.CreateException {
       setStreet(street);
```

```
    setCity(city);
    setState(state);
    setZip(zip);
    return null;
}
```

The Enterprise Bean Class

The EJB implementation class implements the bean's business methods of the component interface and the methods dedicated to the EJB environment, the interface of which is explicitly defined in the EJB specification. The class must implement the `javax.ejb.EntityBean` interface, must be defined as public, cannot be *abstract* for CMP 1.1, and must be abstract for CMP 2.0 (in this case, the abstract methods are the get and set accessor methods of the bean cmp and cmr fields). Following is a list of the EJB environment dedicated methods that the EJB provider must develop.

The first set of methods are those corresponding to the create and find methods of the Home interface:

- *public PrimaryKeyClass* ***ejbCreate****(...);*

  This method is invoked by the container when a client invokes the corresponding create operation on the enterprise Bean's home interface. The method should initialize instance's variables from the input arguments. The returned object should be the primary key of the created instance. For bean−managed persistence, the bean provider should develop here the JDBC code to create the corresponding data in the database. For container−managed persistence, the container will perform the database insert **after** the ejbCreate method completes and the return value should be *null*.
- *public void* ***ejbPostCreate****(...);*

  There is a matching ejbPostCreate method (same input parameters) for each ejbCreate method. The container invokes this method after the execution of the matching ejbCreate(...) method. During the ejbPostCreate method, the object identity is available.
- *public* <PrimaryKeyClass or Collection> ***ejbFind<method>*** *(...); // bean−managed persistence only*

  The container invokes this method on a bean instance that is not associated with any particular object identity (some kind of class method ...) when the client invokes the corresponding method on the Home interface. The implementation uses the arguments to locate the requested object(s) in the database and returns a primary key (or a collection thereof). Currently, collections will be represented as `java.util.Enumeration objects` or `java.util.Collection`. The mandatory *FindByPrimaryKey* method takes as argument a primary key type value and returns a primary key object (it verifies that the corresponding entity exists in the database). **For container−managed persistence**, the bean provider does not have to write these finder methods; they are generated at deployment time by the EJB platform tools. The information needed by the EJB platform for automatically generating these finder methods should be provided by the bean programmer. The EJB 1.1 specification does not specify the format of this finder method description; for *JOnAS*, the CMP 1.1 finder methods description should be provided in the JOnAS−specific deployment descriptor of the Entity Bean (as an SQL query). Refer to the section "Configuring database access for container−managed

persistence." The EJB 2.0 specification defines a standard way to describe these finder methods, i.e. in the standard deployment descriptor, as an EJB−QL query. Also refer to the section "Configuring database access for container−managed persistence." Then, the methods of the `javax.ejb.EntityBean` interface must be implemented:

- *public void* ***setEntityContext****(EntityContext ic);*

  Used by the container to pass a reference to the EntityContext to the bean instance. The container invokes this method on an instance after the instance has been created. Generally, this method is used to store this reference in an instance variable.
- *public void* ***unSetEntityContext****();*

  Unset the associated entity context. The container calls this method before removing the instance. This is the last method the container invokes on the instance.
- *public void* ***ejbActivate****();*

  The container invokes this method when the instance is taken out of the pool of available instances to become associated with a specific EJB object. This method transitions the instance to the ready state.
- *public void* ***ejbPassivate****();*

  The container invokes this method on an instance before the instance becomes dissociated with a specific EJB object. After this method completes, the container will place the instance into the pool of available instances.
- *public void* ***ejbRemove****();*

  This method is invoked by the container when a client invokes a remove operation on the enterprise bean. For entity beans with *bean−managed persistence*, this method should contain the JDBC code to remove the corresponding data in the database. For *container−managed persistence*, this method is called **before** the container removes the entity representation in the database.
- *public void* ***ejbLoad****();*

  The container invokes this method to instruct the instance to synchronize its state by loading it from the underlying database. For *bean−managed persistence*, the EJB provider should code at this location the JDBC statements for reading the data in the database. For *container−managed persistence*, loading the data from the database will be done automatically by the container just **before** ejbLoad is called, and the ejbLoad method should only contain some "after loading calculation statements."
- *public void* ***ejbStore****();*

  The container invokes this method to instruct the instance to synchronize its state by storing it to the underlying database. For *bean−managed persistence*, the EJB provider should code at this location the JDBC statements for writing the data in the database. For entity beans with *container−managed persistence*, this method should only contain some "pre−store statements," since the container will extract the container−managed fields and write them to the database just **after** the ejbStore method call.

Example

The following examples are for container−managed persistence with EJB 1.1 and EJB 2.0. For bean−managed persistence, refer to the examples delivered with your specific platform.

CMP 1.1

```
package eb;

import java.rmi.RemoteException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import javax.ejb.ObjectNotFoundException;
import javax.ejb.RemoveException;
import javax.ejb.EJBException;

public class AccountImplBean implements EntityBean {

    // Keep the reference on the EntityContext
    protected EntityContext entityContext;

    // Object state
    public Integer accno;
    public String customer;
    public double balance;

    public Integer ejbCreate(int val_accno, String val_customer, double val_balance) {

        // Init object state
        accno = new Integer(val_accno);
        customer = val_customer;
        balance = val_balance;
        return null;
    }

    public void ejbPostCreate(int val_accno, String val_customer, double val_balance) {
        // Nothing to be done for this simple example.
    }

    public void ejbActivate() {
        // Nothing to be done for this simple example.
    }

    public void ejbLoad() {
        // Nothing to be done for this simple example, in implicit persistence.
    }

    public void ejbPassivate() {
        // Nothing to be done for this simple example.
    }


    public void ejbRemove() {
```

```
        // Nothing to be done for this simple example, in implicit persistence.
    }

    public void ejbStore() {
        // Nothing to be done for this simple example, in implicit persistence.
    }

    public void setEntityContext(EntityContext ctx) {
        // Keep the entity context in object
        entityContext = ctx;
    }

    public void unsetEntityContext() {
        entityContext = null;
    }

    public double getBalance() {
        return balance;
    }

    public void setBalance(double d) {
        balance = balance + d;
    }

    public String  getCustomer() {
        return customer;
    }

    public void setCustomer(String c) {
        customer = c;
    }

    public int getNumber()  {
        return accno.intValue();
    }
}
```

CMP 2.0

```
import java.rmi.RemoteException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import javax.ejb.ObjectNotFoundException;
import javax.ejb.RemoveException;
import javax.ejb.CreateException;
import javax.ejb.EJBException;

public abstract class AccountImpl2Bean implements EntityBean {

    // Keep the reference on the EntityContext
    protected EntityContext entityContext;


    /*======================== Abstract set and get accessors for cmp fields ==============*/
```

```
public abstract String getCustomer();
public abstract void setCustomer(String customer);

public abstract double getBalance();
public abstract void setBalance(double balance);

public abstract int getAccno();
public abstract void setAccno(int accno);

/*========================= ejbCreate methods ============================*/


public Integer ejbCreate(int val_accno, String val_customer, double val_balance)
    throws CreateException {

    // Init object state
    setAccno(val_accno);
    setCustomer(val_customer);
    setBalance(val_balance);
    return null;
}

public void ejbPostCreate(int val_accno, String val_customer, double val_balance) {
    // Nothing to be done for this simple example.
}


/*====================== javax.ejb.EntityBean implementation =================*/

public void ejbActivate() {
    // Nothing to be done for this simple example.
}

public void ejbLoad() {
    // Nothing to be done for this simple example, in implicit persistence.
}

public void ejbPassivate() {
    // Nothing to be done for this simple example.
}

public void ejbRemove() throws RemoveException {
    // Nothing to be done for this simple example, in implicit persistence.
}

public void ejbStore() {
    // Nothing to be done for this simple example, in implicit persistence.
}

public void setEntityContext(EntityContext ctx) {

    // Keep the entity context in object
    entityContext = ctx;
```

```
    }

    public void unsetEntityContext()  {
        entityContext = null;
    }

    /**
     * Business method to get the Account number
     */
    public int getNumber()  {
        return getAccno();
    }

}
```

Writing Database Access Operations (bean−managed persistence)

For *bean−managed persistence*, data access operations are developed by the bean provider using the JDBC interface. However, getting database connections must be obtained through the *javax.sql.DataSource* interface on a datasource object provided by the EJB platform. This is mandatory since the EJB platform is responsible for managing the connection pool and for transaction management. Thus, to get a JDBC connection, in each method performing database operations, the bean provider must:

- call the **getConnection(...)** method of the DataSource object, to obtain a connection to perform the JDBC operations in the current transactional context (if there are JDBC operations),
- call the **close()** method on this connection after the database access operations, so that the connection can be returned to the connection pool (and be dissociated from the potential current transaction).

A method that performs database access must always contain the getConnection and close statements, as follows:

```
public void doSomethingInDB (...) {
    conn = dataSource.getConnection();
    ... // Database access operations
    conn.close();
}
```

A DataSource object associates a JDBC driver with a database (as an ODBC datasource). It is created and registered in JNDI by the EJB server at launch time (refer also to the section "JDBC DataSources configuration").

A DataSource object is a resource manager connection factory for `java.sql.Connection` objects, which implements connections to a database management system. The enterprise bean code refers to resource factories using logical names called "Resource manager connection factory references." The resource manager connection factory references are special entries in the enterprise bean environment. The bean provider must use resource manager connection factory references to obtain the datasource object as follow:

- Declare the resource reference in the standard deployment descriptor using a `resource−ref` element.
- Lookup the datasource in the enterprise bean environment using the JNDI interface (refer to the section "Enterprise Bean's Environment").

The deployer binds the resource manager connection factory references to the actual resource factories that are configured in the server. This binding is done in the JOnAS−specific deployment descriptor using the jonas−resource element.

Example

The declaration of the resource reference in the standard deployment descriptor looks like the following:

```
<resource−ref>
<res−ref−name>jdbc/AccountExplDs</res−ref−name>
<res−type>javax.sql.DataSource</res−type>
<res−auth>Container</res−auth>
</resource−ref>
```

The <res−auth> element indicates which of the two resource manager authentication approaches is used:

- *Container*: the deployer sets up the sign−on information.
- *Bean*: the bean programmer should use the getConnection method with user and password parameters.

The JOnAS−specific deployment descriptor must map the environment JNDI name of the resource to the actual JNDI name of the resource object managed by the EJB server. This is done in the <jonas−resource> element.

```
  <jonas−entity>
    <ejb−name>AccountExpl</ejb−name>
    <jndi−name>AccountExplHome</jndi−name>
    <jonas−resource>
      <res−ref−name>jdbc/AccountExplDs</res−ref−name>
      <jndi−name>jdbc_1</jndi−name>
    </jonas−resource>
  </jonas−entity>
```

The ejbStore method of the same Account example with bean−managed persistence is shown in the following example. It performs JDBC operations to update the database record representing the state of the entity bean instance. The JDBC connection is obtained from the datasource associated with the bean. This datasource has been instantiated by the EJB server and is available for the bean through its resource reference name, which is defined in the standard deployment descriptor.

In the bean, a reference to a datasource object of the EJB server is initialized:

```
it = new InitialContext();

ds = (DataSource)it.lookup("java:comp/env/jdbc/AccountExplDs");
```

Then, this datasource object is used in the implementation of the methods performing JDBC operations, such as ejbStore, as illustrated in the following:

```
public void ejbStore
```

```
    Connection conn = null;
    PreparedStatement stmt = null;
    try { // get a connection
        conn = ds.getConnection();
        // store Object state in DB
        stmt = conn.prepareStatement("update account set customer=?,balance=? where accno=?");
        stmt.setString(1, customer);
        stmt.setDouble(2, balance);
        Integer pk = (Integer)entityContext.getPrimaryKey();
        stmt.setInt(3, pk.accno);
        stmt.executeUpdate();
     } catch (SQLException e) {
        throw new javax.ejb.EJBException("Failed to store bean to database", e);
     } finally {
        try {
            if (stmt != null) stmt.close();    // close statement
            if (conn != null) conn.close();    // release connection
        } catch (Exception ignore) {}
     }
}
```

Note that the close statement instruction may be important if the server is intensively accessed by many clients performing entity bean access. If the statement is not closed in the finally block, since *stmt* is in the scope of the method, it will be deleted at the end of the method (and the close will be implicitly done). However, it may be some time before the Java garbage collector deletes the statement object. Therefore, if the number of clients performing entity bean access is important, the DBMS may raise a "too many opened cursors" exception (a JDBC statement corresponds to a DBMS cursor). Since connection pooling is performed by the platform, closing the connection will not result in a physical connection close, therefore opened cursors will not be closed. Thus, it is preferable to explicitly close the statement in the method.

It is a good programming practice to put the JDBC connection and JDBC statement close operations in a finally bloc of the try statement.

Configuring Database Access for Container−managed Persistence

The standard way to indicate to an EJB platform that an entity bean has container−managed persistence is to fill the `<persistence−type>` tag of the deployment descriptor with the value "container," and to fill the `<cmp−field>` tag of the deployment descriptor with the list of container−managed fields (the fields that the container will have in charge to make persistent). The CMP version (1.x or 2.x) should also be specified in the <cmp−version> tag. In the textual format of the deployment descriptor, this is represented by the following lines:

```
    <persistence−type>container</persistence−type>
    <cmp−version>1.x</cmp−version>
    <cmp−field>
      <field−name>fieldOne</field−name>
    </cmp−field>
    <cmp−field>
      <field−name>fieldTwo</field−name>
    </cmp−field>
```

With *container−managed persistence* the programmer need not develop the code for accessing the data in the relational database; this code is included in the container itself (generated by the platform tools). However, for the EJB platform to know how to access the database and which data to read and write in the database, two types of information must be provided with the bean:

- First, the container must know which database to access and how to access it. To do this, the only required information is the **name of the DataSource** that will be used to get the JDBC connection. For container−managed persistence, only one DataSource per bean should be used.
- Then, it is necessary to know **the mapping of the bean fields to the underlying database** (which table, which column). For CMP 1.1 or CMP 2.0, this mapping is specified by the deployer in the JOnAS−specific deployment descriptor. Note that for CMP 2.0, this mapping may be entirely generated by JOnAS.

The EJB specification does not specify how this information should be provided to the EJB platform by the bean deployer. Therefore, what is described in the remainder of this section is *specific to JOnAS*.

For CMP 1.1, the bean deployer is responsible for defining the mapping of the bean fields to the database table columns. The name of the DataSource can be set at deployment time, since it depends on the EJB platform configuration. This database configuration information is defined in the JOnAS−specific deployment descriptor via the `jdbc−mapping` element. The following example defines the mapping for a CMP 1.1 entity bean:

```
<jdbc−mapping>
  <jndi−name>jdbc_1</jndi−name>
  <jdbc−table−name>accountsample</jdbc−table−name>
  <cmp−field−jdbc−mapping>
  <field−name>mAccno</field−name>
  <jdbc−field−name>accno</jdbc−field−name>
  </cmp−field−jdbc−mapping>
  <cmp−field−jdbc−mapping>
  <field−name>mCustomer</field−name>
  <jdbc−field−name>customer</jdbc−field−name>
  </cmp−field−jdbc−mapping>
  <cmp−field−jdbc−mapping>
  <field−name>mBalance</field−name>
  <jdbc−field−name>balance</jdbc−field−name>
</jdbc−mapping>
```

`jdbc_1` is the JNDI name of the DataSource object identifying the database. `accountsample` is the name of the table used to store the bean instances in the database. `mAccno`, `mCustomer`, and `mBalance` are the names of the container−managed fields of the bean to be stored in the `accno`, `customer`, and `balance` columns of the `accountsample` table. This example applies to *container−managed persistence*. For *bean−managed persistence*, the database mapping does not exist.

For a CMP 2.0 entity bean, only the `jndi−name` element of the `jdbc−mapping` is mandatory, since the mapping may be generated automatically (for an explicit mapping definition, refer to the "JOnAS Database Mapping" section of the Using CMP2.0 persistence chapter):

```
<jdbc−mapping>
  <jndi−name>jdbc_1</jndi−name>
```

```
    </jdbc-mapping>
    <cleanup>create</cleanup>
```

For a CMP 2.0 entity bean, the JOnAS-specific deployment descriptor contains an additional element, `cleanup`, at the same level as the `jdbc-mapping` element, which can have one of the following values:

removedata
: at bean loading time, the content of the tables storing the bean data is deleted

removeall
: at bean loading time, the tables storing the bean data are dropped (if they exist) and created

none
: do nothing

create
: default value (if the element is not specified), at bean loading time, the tables for storing the bean data are created if they do not exist

For CMP 1.1, the `jdbc-mapping` element can also contain information defining the behaviour of the implementation of a *find<method>* method (i.e. the *ejbFind<method>* method, that will be generated by the platform tools). This information is represented by the `finder-method-jdbc-mapping` element.

For each finder method, this element provides a way to define an SQL WHERE clause that will be used in the generated finder method implementation to query the relational table storing the bean entities. Note that the table column names should be used, not the bean field names. Example:

```
<finder-method-jdbc-mapping>
  <jonas-method>
    <method-name>findLargeAccounts</method-name>
  </jonas-method>
  <jdbc-where-clause>where balance &gt; ?</jdbc-where-clause>
</finder-method-jdbc-mapping>
```

The previous finder method description will cause the platform tools to generate an implementation of ejbFindLargeAccount(double arg) that returns the primary keys of the entity bean objects corresponding to the tuples returned by the "select ... from Account where balance > ?", where '?' will be replaced by the value of the first argument of the findLargeAccount method. If several '?' characters appear in the provided WHERE clause, this means that the finder method has several arguments and the '?' characters will correspond to these arguments, adhering to the order of the method signature.

In the WHERE clause, the parameters can be followed by a number, which specifies the method parameter number that will be used by the query in this position.
Example: The WHERE clause of the following finder method can be:

```
Enumeration findByTextAndDateCondition(String text, java.sql.Date date)

WHERE (description like ?1 OR summary like ?1) AND (?2 &gt; date)
```

Note that a `<finder−method−jdbc−mapping>` element for the *findByPrimaryKey* method is not necessary, since the meaning of this method is known.

Additionally, note that for CMP 2.0, the information defining the behaviour of the implementation of a *find<method>* method is located in **the standard deployment descriptor**, as an EJB−QL query (i.e. this is not JOnAS−specific information). The same finder method example in CMP 2.0:

```
<query>
  <query−method>
    <method−name>findLargeAccounts</method−name>
    <method−params>
        <method−param>double</method−param>
    </method−params>
  </query−method>
  <ejb−ql>SELECT OBJECT(o) FROM accountsample o WHERE o.balance &gt; ?1</ejb−ql>
</query>
```

The datatypes supported for container−managed fields in CMP 1.1 are the following:

Java Type	**JDBC Type**	**JDBC driver Access methods**
boolean	BIT	getBoolean(), setBoolean()
byte	TINYINT	getByte(), setByte()
short	SMALLINT	getShort(), setShort()
int	INTEGER	getInt(), setInt()
long	BIGINT	getLong(), setLong()
float	FLOAT	getFloat(), setFloat()
double	DOUBLE	getDouble(), setDouble
byte[]	VARBINARY or LONGVARBINARY (1)	getBytes(), setBytes()
java.lang.String	VARCHAR or LONGVARCHAR (1)	getString(), setString()
java.lang.Boolean	BIT	getBoolean(), setObject()
java.lang.Integer	INTEGER	getInt(), setObject()
java.lang.Short	SMALLINT	getShort(), setObject()
java.lang.Long	BIGINT	getLong(), setObject()
java.lang.Float	REAL	getFloat(), setObject()
java.lang.Double	DOUBLE	getDouble(), setObject()
java.math.BigDecimal	NUMERIC	getBigDecimal(),

		setObject()
java.math.BigInteger	NUMERIC	getBigDecimal(), setObject()
java.sql.Date	DATE	getDate(), setDate()
java.sql.Time	TIME	getTime(), setTime()
java.sql.Timestamp	TIMESTAMP	getTimestamp(), setTimestamp()
any serializable class	VARBINARY or LONGVARBINARY (1)	getBytes(), setBytes()

(1) The mapping for String will normally be VARCHAR, but will turn into LONGVARCHAR if the given value exceeds the driver's limit on VARCHAR values. The case is similar for byte[] and VARBINARY and LONGVARBINARY values.

For CMP 2.0, the supported datatypes depend on the JORM mapper used.

Tuning Container for Entity Bean Optimizations

JOnAS must make a compromise between scalability and performance. Towards this end, we have introduced five new tags in the JOnAS−specific deployment descriptor:

shared

This optional flag must be defined as `True,` if the bean−persistent state can be accessed outside the JOnAS Server. When this flag is `False`, the JOnAS Server can do some optimization, such as not re−reading the bean state before starting a new transaction.

min−pool−size

This optional integer value represents the minimum instances that will be created in the pool when the bean is loaded. This will improve bean instance create time, at least for the first instances.

max−cache−size

This optional integer value represents the maximum of instances in memory. The purpose of this value is to keep JOnAS scalable.

is−modified−method−name

To improve performance of CMP 1.1 entity beans, JOnAS implements the isModified extension. Before performing an update, the container calls a method of the bean whose name is identified in the `is−modified−method−name` element of the JOnAS−specific deployment descriptor. This method is responsible for determining if the state of the bean has been changed. By doing this, the container determines if it must store data in the database or not.

Example

The bean implementation manages a boolean `isDirty` and implements a method that returns the value of this boolean: `isModified`

```
private transient boolean isDirty;
public boolean isModified() {
    return isDirty;
}
```

The JOnAS−specific deployment descriptor directs the bean to implement an `isModified` method:

```
<jonas−entity>
  <ejb−name>Item</ejb−name>
  <is−modified−method−name>isModified</is−modified−method−name>
  .....
</jonas−entity>
```

Methods that modify the value of the bean must set the flag `isDirty` to `true`.
Methods that restore the value of the bean from the database must reset the flag `isDirty` to `false`. Therefore, the flag must be set to `false` in the `ejbLoad()` and `ejbStore()` methods.

passivation−timeout

Entity bean instances are passivated at the end of the transaction and reactivated at the beginning of the next transaction. In the event that these instances are accessed outside a transaction, their state is kept in memory to improve performance. However, a passivation will occur in three situations:

1. When the bean is unloaded from the server, at a minimum when the server is stopped.
2. When a transaction is started on this instance.
3. After a configurable timeout. If the bean is always accessed with no transaction, it may be prudent to periodically store the bean state on disk.

This passivation timeout can be configured in the JOnAS−specific deployment descriptor, with a non−mandatory tag <passivation−timeout>. Example:

```
<jonas−entity>
  <ejb−name>Item</ejb−name>
  <passivation−timeout>5</passivation−timeout>
  .....
</jonas−entity>
```

This entity bean will be passivated every five second, if not accessed within transactions.

Using CMP2.0 persistence

This section highlights the main differences between CMP as defined in EJB 2.0 specification (called CMP2.0) and CMP as defined in EJB 1.1 specification (called CMP1.1). Major new features in the standard development and deployment of CMP2.0 entity beans are listed (comparing them to CMP1.1), along with JOnAS−specific information. Mapping CMP2.0 entity beans to the database is described in detail. Note that the database mapping can be created entirely by JOnAS, in which case the JOnAS−specific deployment descriptor for an entity bean should contain only the datasource and the element indicating how the database should be initialized.

Standard CMP2.0 Aspects

This section briefly describes the new features available in CMP2.0 as compared to CMP 1.1, and how these features change the development of entity beans.

Entity Bean Implementation Class

The EJB implementation class 1) implements the bean's business methods of the component interface, 2) implements the methods dedicated to the EJB environment (the interface of which is explicitly defined in the EJB specification), and 3) defines the abstract methods representing both the persistent fields (cmp−fields) and the relationship fields (cmr−fields). The class must implement the `javax.ejb.EntityBean` interface, be defined as public, and be abstract (which is not the case for CMP1.1, where it must not be abstract). The abstract methods are the get and set accessor methods of the bean cmp and cmr fields. Refer to the examples and details in the section "Developing Entity Beans" of the JOnAS documentation.

Standard Deployment Descriptor

The standard way to indicate to an EJB platform that an entity bean has container−managed persistence is to fill the `<persistence−type>` tag of the deployment descriptor with the value "container," and to fill the `<cmp−field>` tags of the deployment descriptor with the list of container−managed fields (the fields that the container will have in charge to make persistent) and the `<cmr−field>` tags identifying the relationships. The CMP version (1.x or 2.x) should also be specified in the <cmp−version> tag. This is represented by the following lines in the deployment descriptor:

```
<persistence−type>container</persistence−type>
<cmp−version>1.x</cmp−version>
<cmp−field>
  <field−name>fieldOne</field−name>
</cmp−field>
<cmp−field>
  <field−name>fieldTwo</field−name>
</cmp−field>
```

Note that for running CMP1.1−defined entity beans on an EJB2.0 platform, such as JOnAS 3.x, ***you must introduce this*** <cmp−version> ***element in your deployment descriptors, since the default cmp−version value (if not specified) is 2.x***.

Note that for CMP 2.0, the information defining the behaviour of the implementation of a *find<method>* method is located in ***the standard deployment descriptor*** as an EJB−QL query (this is not JOnAS−specific information). For CMP 1.1, this information is located in the JOnAS−specific deployment descriptor as an SQL WHERE clause specified in a <finder−method−jdbc−mapping> element.

Finder method example in CMP 2.0: for a `findLargeAccounts(double val)` method defined on the Account entity bean of the JOnAS eb example.

```
<query>
  <query-method>
    <method-name>findLargeAccounts</method-name>
    <method-params>
        <method-param>double</method-param>
    </method-params>
  </query-method>
  <ejb-ql>SELECT OBJECT(o) FROM accountsample o WHERE o.balance &gt; ?1</ejb-ql>
</query>
```

JOnAS Database mappers

For implementing the EJB 2.0 persistence (CMP2.0), JOnAS relies on the JORM framework. JORM itself relies on JOnAS DataSources (specified in DataSource properties files) for connecting to the actual database. JORM must adapt its object−relational mapping to the underlying database, for which it makes use of adapters called "mappers." Thus, for each type of database (and more precisely for each JDBC driver), the corresponding mapper must be specified in the DataSource. This is the purpose of the ***datasource.mapper*** property of the DataSource properties file. Note that all JOnAS−provided DataSource properties files (in JOnAS_ROOT/conf) already contain this property with the correct mapper.

property name	description	possible values
datasource.mapper	JORM database mapper	• `rdb`: generic mapper (JDBC standard driver ...) • `rdb.firebird`: Firebird • `rdb.mckoi`: McKoi Db • `rdb.mysql`: MySQL • `rdb.oracle8`: Oracle 8 and lesser versions • `rdb.oracle`: Oracle 9 • `rdb.postgres`: PostgreSQL (>= 7.2) • `rdb.sapdb`: Sap DB • `rdb.sqlserver`: MS Sql Server • `rdb.sybase`: Sybase

Contact the JOnAS team to obtain a mapper for other databases.

The container code generated at deployment (GenIC or ejbjar ant task) is dependent on this mapper. It is possible to deploy (generate container code) a bean for several mappers in order to change the database (i.e. the DataSource file) without redeploying the bean. These mappers should be specified as the ***mappernames*** argument of the GenIC command or as the ***mappernames*** attribute of the JOnAS ANT ejbjar task. The value is a comma−separated list of mapper names for which the container classes will be generated. This list of mapper names corresponds to the list of potential databases upon which you can deploy your entity beans. For example, to deploy entity beans so that they may be used on either Oracle or PostgreSQL, run GenIC as:

```
GenIC −mappernames rdb.oracle,rdb.postgres eb.jar
```

The following is the same example in an ANT build.xml file:

```
<target name="deploy"
        description="Build and deploy the ejb−jars"
        depends="compile>
  <ejbjar naming="directory"
      ....
      ....
    <jonas destdir="${ejbjars.dir}"
           jonasroot="${jonas.root}"
           orb="${orb}"
           jarsuffix=".jar"
           secpropag="yes"
           keepgenerated="true"
           mappernames="${mapper.names}"
           additionalargs="${genicargs}">
    </jonas>
    ...
    ...
  </ejbjar>
</target>
```

having in build.properties:

```
# mappers for entity CMP2
mapper.names            rdb.oracle,rdb.postgres
```

JOnAS Database Mapping (Specific Deployment Descriptor)

The mapping to the database of entity beans and their relationships may be specified in the JOnAS−specific deployment descriptor, in `jonas−entity` elements, and in `jonas−ejb−relation` elements. Since JOnAS is able to generate the database mapping, all the elements of the JOnAS−specific deployment descriptor defined in this section (which are sub−elements of `jonas−entity` or `jonas−ejb−relation`) are optional, except those for specifying the datasource and the initialization mode (i.e. the `jndi−name` of `jdbc−mapping` and `cleanup`). The default values of these mapping elements, provided in this section, define the JOnAS−generated database mapping.

Specifying and Initializing the Database

For specifying the database within which a CMP 2.0 entity bean is stored, the `jndi-name` element of the `jdbc-mapping` is necessary. This is the JNDI name of the DataSource representing the database storing the entity bean.

```
<jdbc-mapping>
  <jndi-name>jdbc_1</jndi-name>
</jdbc-mapping>
```

For a CMP 2.0 entity bean, the JOnAS-specific deployment descriptor contains an additional element, `cleanup`, to be specified before the `jdbc-mapping` element, which can have one of the following values:

removedata
: at bean loading time, the content of the tables storing the bean data is deleted

removeall
: at bean loading time, the tables storing the bean data are dropped (if they exist) and created

none
: do nothing

create
: default value (if the element is not specified), at bean loading time, the tables for storing the bean data are created if they do not exist.

It may be useful for testing purposes to delete the database data each time a bean is loaded. For this purpose, the part of the JOnAS-specific deployment descriptor related to the entity bean may look like the following:

```
<cleanup>removedata</cleanup>
<jdbc-mapping>
  <jndi-name>jdbc_1</jndi-name>
</jdbc-mapping>
```

CMP fields Mapping

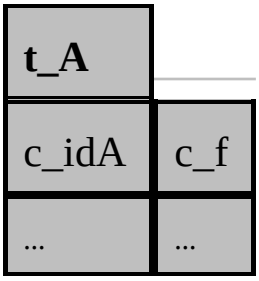
Mapping CMP fields in CMP2.0 is similar to that of CMP 1.1, but in CMP2.0 it is also possible to specify the SQL type of a column. Usually this SQL type is used if JOnAS creates the table (`create` value of the `cleanup` element), and if the JORM default chosen SQL type is not appropriate.

Standard Deployment Descriptor

```
.....
<entity>
  <ejb-name>A</ejb-name>
  .....
  <cmp-field>
    <field-name>idA</field-name>
  </cmp-field>
```

```
    <cmp-field>
      <field-name>f</field-name>
    </cmp-field>
  .....
  </entity>
  .....
```

Database Mapping



JOnAS Deployment Descriptor

```
  .....
  <jonas-entity>
    <ejb-name>A</ejb-name>
    .....
    <jdbc-mapping>
      <jndi-name>jdbc_1</jndi-name>
      <jdbc-table-name>t_A</jdbc-table-name>
      <cmp-field-jdbc-mapping>
        <field-name>idA</field-name>
        <jdbc-field-name>c_idA</jdbc-field-name>
      </cmp-field-jdbc-mapping>
      <cmp-field-jdbc-mapping>
        <field-name>f</field-name>
        <jdbc-field-name>c_f</jdbc-field-name>
        <sql-type>varchar(40)</sql-type>
      </cmp-field-jdbc-mapping>
    </jdbc-mapping>
    .....
  </jonas-entity>
  .....
```

Defaults values:

jndi-name	Mandatory
jdbc-table-name	Optional. Default value is the *upper-case* CMP2 abstract-schema-name, or the CMP1 ejb-name, suffixed by _.
cmp-field-jdbc-mapping	Optional.
jdbc-field-name	

	Optional. Default value is the field−name suffixed by **_**. **idA_** and **f_** in the example.
sql−type	Optional. Default value defined by JORM.

CMR fields Mapping to primary−key−fields (simple pk)

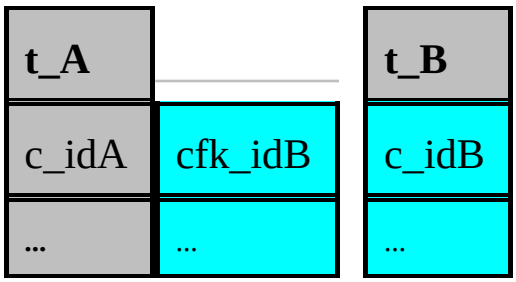
1−1 unidirectional relationships

Standard Deployment Descriptor

```
.....
<entity>
  <ejb−name>A</ejb−name>
  .....
  <cmp−field>
    <field−name>idA</field−name>
  </cmp−field>
  <primkey−field>idA</primkey−field>
  .....
</entity>
.....
<entity>
  <ejb−name>B</ejb−name>
  .....
  <cmp−field>
    <field−name>idB</field−name>
  </cmp−field>
  <primkey−field>idB</primkey−field>
  .....
</entity>
.....
<relationships>
  <ejb−relation>
    <ejb−relation−name>a−b</ejb−relation−name>
    <ejb−relationship−role>
      <!−− A => B −−>
            <ejb−relationship−role−name>a2b</ejb−relationship−role−name>
      <multiplicity>One</multiplicity>
      <relationship−role−source>
        <ejb−name>A</ejb−name>
      </relationship−role−source>
      <cmr−field>
        <cmr−field−name>b</cmr−field−name>
      </cmr−field>
    </ejb−relationship−role>
    <ejb−relationship−role>
      <!−− B => A −−>
```

```
      <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>B</ejb-name>
      </relationship-role-source>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
.....
```

Database Mapping



There is a foreign key in the table of the bean that owns the CMR field.

JOnAS Deployment Descriptor

```
.....
<jonas-entity>
  <ejb-name>A</ejb-name>
  .....
  <jdbc-mapping>
    <jndi-name>jdbc_1</jndi-name>
    <jdbc-table-name>t_A/jdbc-table-name>
    <cmp-field-jdbc-mapping>
      <field-name>idA</field-name>
      <jdbc-field-name>c_idA</jdbc-field-name>
    </cmp-field-jdbc-mapping>
  </jdbc-mapping>
  .....
</jonas-entity>
.....
<jonas-entity>
  <ejb-name>B</ejb-name>
  .....
  <jdbc-mapping>
    <jndi-name>jdbc_1</jndi-name>
    <jdbc-table-name>t_B/jdbc-table-name>
    <cmp-field-jdbc-mapping>
      <field-name>idB</field-name>
      <jdbc-field-name>c_idB</jdbc-field-name>
    </cmp-field-jdbc-mapping>
  </jdbc-mapping>
  .....
```

```
</jonas-entity>
.....
<jonas-ejb-relation>
  <ejb-relation-name>a-b</ejb-relation-name>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_idb</foreign-key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
</jonas-ejb-relation>
.....
```

foreign-key-jdbc-name is the column name of the foreign key in the table of the source bean of the relationship-role.
In this example, where the destination bean has a primary-key-field, it is possible to deduce that this foreign-key-jdbc-name column is to be associated with the column of this primary-key-field in the table of the destination bean.

Default values:

jonas-ejb-relation	Optional
foreign-key-jdbc-name	Optional. Default value is the abstract-schema-name of the destination bean, suffixed by **_**, and by its primary-key-field. **B_idb** in the example.

1-1 bidirectional relationships

Compared to 1-1 unidirectional relationships, there is a CMR field in both of the beans, thus making two types of mapping possible.

Standard Deployment Descriptor

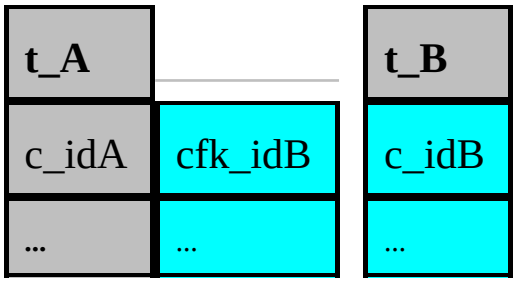
```
.....
<relationships>
  <ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
    <ejb-relationship-role>
      <!-- A => B -->
            <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>A</ejb-name>
      </relationship-role-source>
      <cmr-field>
```

```
        <cmr-field-name>b</cmr-field-name>
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <!-- B => A -->
      <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>B</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>a</cmr-field-name>
      </cmr-field>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
.....
```

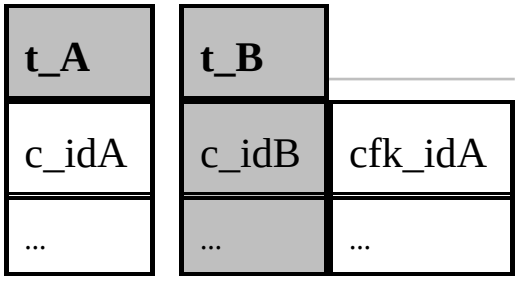
Database Mapping

Two mappings are possible. One of the tables may hold a foreign key.

Case 1:



Case 2:



JOnAS Deployment Descriptor

Case 1:

```
.....
<jonas-ejb-relation>
  <ejb-relation-name>a-b</ejb-relation-name>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_idb</foreign-key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
</jonas-ejb-relation>
.....
```

Case 2:

```
.....
<jonas-ejb-relation>
  <ejb-relation-name>a-b</ejb-relation-name>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_ida</foreign-key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
</jonas-ejb-relation>
.....
```

For the default mapping, the foreign key is in the table of the source bean of the first ejb−relationship−role of the ejb−relation. In the example, the default mapping corresponds to case 1, since the ejb−relationship−role *a2b* is the first defined in the ejb−relation *a−b*. Then, the default values are similar to those of the 1−1 unidirectional relationship.

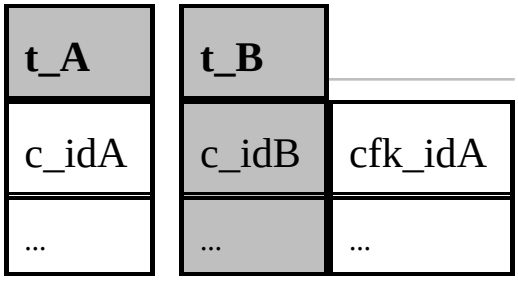
1−N unidirectional relationships

Standard Deployment Descriptor

```
.....
<relationships>
  <ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
    <ejb-relationship-role>
      <!-- A => B -->
            <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>A</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>b</cmr-field-name>
        <cmr-field-type>java.util.Collection</cmr-field-type>
```

```
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <!-- B => A -->
      <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>B</ejb-name>
      </relationship-role-source>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
.....
```

Database Mapping



In this case, the foreign key must be in the table of the bean which is on the "many" side of the relationship (i.e. in the table of the source bean of the relationship role with multiplicity many), t_B.

JOnAS Deployment Descriptor

```
.....
<jonas-ejb-relation>
  <ejb-relation-name>a-b</ejb-relation-name>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_ida</foreign-key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
</jonas-ejb-relation>
.....
```

Default values:

jonas-ejb-relation	Optional
foreign-key-jdbc-name	Optional. Default value is the abstract-schema-name of the destination bean of the "one" side of the relationship (i.e. the source bean of the relationship role with multiplicity one), suffixed by _, and by its primary-key-field.

	A_ida in the example.

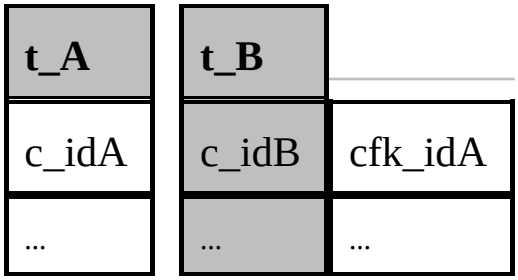
1−N bidirectional relationships

Similar to 1−N unidirectional relationships, but with a CMR field in each bean.

Standard Deployment Descriptor

```
.....
<relationships>
  <ejb−relation>
    <ejb−relation−name>a−b</ejb−relation−name>
    <ejb−relationship−role>
      <!−− A => B −−>
            <ejb−relationship−role−name>a2b</ejb−relationship−role−name>
      <multiplicity>One</multiplicity>
      <relationship−role−source>
        <ejb−name>A</ejb−name>
      </relationship−role−source>
      <cmr−field>
        <cmr−field−name>b</cmr−field−name>
        <cmr−field−type>java.util.Collection</cmr−field−type>
      </cmr−field>
    </ejb−relationship−role>
    <ejb−relationship−role>
      <!−− B => A −−>
      <ejb−relationship−role−name>b2a</ejb−relationship−role−name>
      <multiplicity>Many</multiplicity>
      <relationship−role−source>
        <ejb−name>B</ejb−name>
      </relationship−role−source>
      <cmr−field>
        <cmr−field−name>a</cmr−field−name>
      </cmr−field>
    </ejb−relationship−role>
  </ejb−relation>
</relationships>
.....
```

Database mapping



In this case, the foreign key must be in the table of the bean which is on the "many" side of the relationship (i.e. in the table of the source bean of the relationship role with multiplicity many), t_B.

JOnAS Deployment Descriptor

```
.....
<jonas-ejb-relation>
  <ejb-relation-name>a-b</ejb-relation-name>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_ida</foreign-key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
</jonas-ejb-relation>
.....
```

Default values:

jonas−ejb−relation	Optional
foreign−key−jdbc−name	Optional. Default value is the abstract−schema−name of the destination bean of the "one" side of the relationship (i.e. the source bean of the relationship role with multiplicity one), suffixed by _, and by its primary−key−field. **A_ida** in the example.

N−1 unidirectional relationships

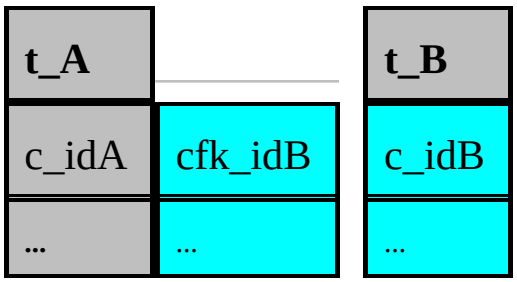
Similar to 1−N unidirectional relationships, but the CMR field is defined on the "many" side of the relationship, i.e. on the (source bean of the) relationship role with multiplicity "many."

Standard Deployment Descriptor

```
.....
<relationships>
  <ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
```

```
    <ejb-relationship-role>
      <!-- A => B -->
            <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>A</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>b</cmr-field-name>
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <!-- B => A -->
      <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>B</ejb-name>
      </relationship-role-source>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
.....
```

Database mapping



In this case, the foreign key must be in the table of the bean which is on the "many" side of the relationship (i.e. in table of the source bean of the relationship role with multiplicity many), t_A.

JOnAS Deployment Descriptor

```
.....
<jonas-ejb-relation>
  <ejb-relation-name>a-b</ejb-relation-name>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_idb</foreign-key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
</jonas-ejb-relation>
.....
```

Default values:

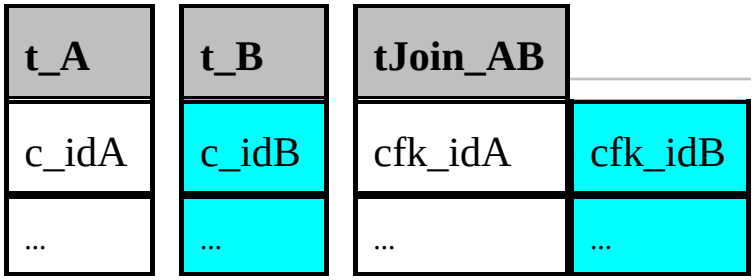
jonas−ejb−relation	Optional
foreign−key−jdbc−name	Optional. Default value is the abstract−schema−name of the destination bean of the "one" side of the relationship (i.e. the source bean of the relationship role with multiplicity one), suffixed by _, and by its primary−key−field. **B_idb** in the example.

N−M unidirectional relationships

Standard Deployment Descriptor

```
.....
<relationships>
  <ejb−relation>
    <ejb−relation−name>a−b</ejb−relation−name>
    <ejb−relationship−role>
      <!−− A => B −−>
            <ejb−relationship−role−name>a2b</ejb−relationship−role−name>
      <multiplicity>Many</multiplicity>
      <relationship−role−source>
        <ejb−name>A</ejb−name>
      </relationship−role−source>
      <cmr−field>
        <cmr−field−name>b</cmr−field−name>
        <cmr−field−type>java.util.Collection</cmr−field−type>
      </cmr−field>
    </ejb−relationship−role>
    <ejb−relationship−role>
      <!−− B => A −−>
      <ejb−relationship−role−name>b2a</ejb−relationship−role−name>
      <multiplicity>Many</multiplicity>
      <relationship−role−source>
        <ejb−name>B</ejb−name>
      </relationship−role−source>
    </ejb−relationship−role>
  </ejb−relation>
</relationships>
.....
```

Database mapping



In this case, there is a join table composed of the foreign keys of each entity bean table.

JOnAS Deployment Descriptor

```
.....
<jonas-ejb-relation>
  <ejb-relation-name>a-b</ejb-relation-name>
  <jdbc-table-name>tJoin_AB</jdbc-table-name>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_idb</foreign-key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_ida</foreign-key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
</jonas-ejb-relation>
.....
```

Default values

jonas−ejb−relation	Optional
jdbc−table−name	Optional. Default value is built from the abstract−schema−names of the beans, separated by **_**. **A_B** in the example.
foreign−key−jdbc−name	Optional. Default value is the abstract−schema−name of the destination bean, suffixed by **_**, and by its primary−key−field. **B_idb** and **A_ida** in the example.

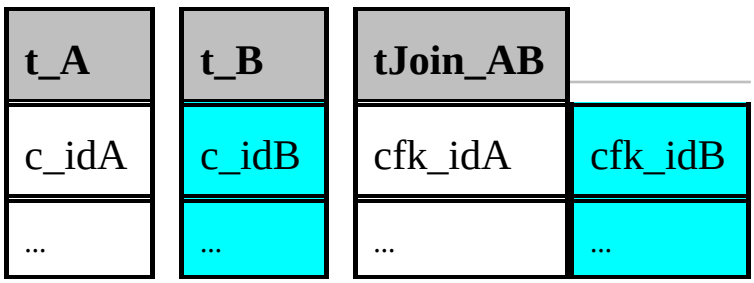
N−M bidirectional relationships

Similar to N−M unidirectional relationships, but a CMR field is defined for each bean.

Standard deployment Descriptor

```
.....
<relationships>
  <ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
    <ejb-relationship-role>
      <!-- A => B -->
            <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>A</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>b</cmr-field-name>
        <cmr-field-type>java.util.Collection</cmr-field-type>
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <!-- B => A -->
      <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>B</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>a</cmr-field-name>
        <cmr-field-type>java.util.Collection</cmr-field-type>
      </cmr-field>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
.....
```

Database mapping



In this case, there is a join table composed of the foreign keys of each entity bean table.

JOnAS Deployment Descriptor

```
.....
<jonas-ejb-relation>
  <ejb-relation-name>a-b</ejb-relation-name>
  <jdbc-table-name>tJoin_AB</jdbc-table-name>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_idb</foreign-key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_ida</foreign-key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
</jonas-ejb-relation>
.....
```

Default values:

jonas-ejb-relation	Optional
jdbc-table-name	Optional. Default value is built from the abstract-schema-names of the beans, separated by **_**. **A_B** in the example.
foreign-key-jdbc-name	Optional. Default value is the abstract-schema-name of the destination bean, suffixed by **_**, and by its primary-key-field. **B_idb** and **A_ida** in the example.

CMR fields Mapping to composite primary-keys

In the case of composite primary keys, the database mapping should provide the capability to specify which column of a foreign key corresponds to which column of the primary key. This is the only difference between relationships based on simple primary keys. For this reason, not all types of relationship are illustrated below.

1-1 bidirectional relationships

Standard Deployment Descriptor

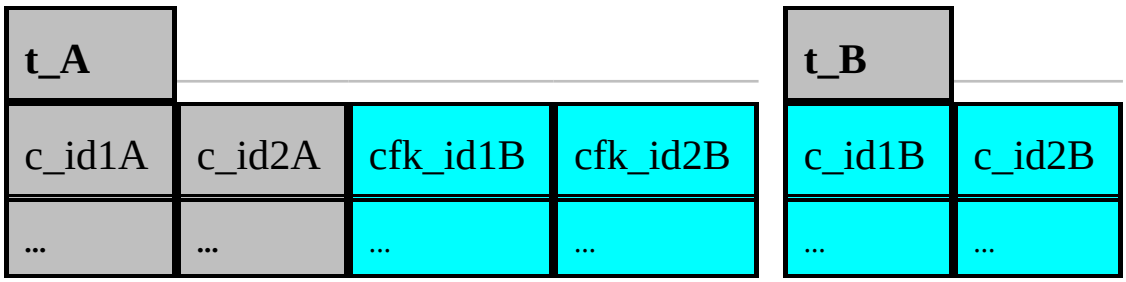
```
.....
<entity>
  <ejb-name>A</ejb-name>
  .....
```

```
  <prim-key-class>p.PkA</prim-key-class>
  .....
  <cmp-field>
    <field-name>id1A</field-name>
  </cmp-field>
  <cmp-field>
    <field-name>id2A</field-name>
  </cmp-field>
  .....
</entity>
.....
<entity>
  <ejb-name>B</ejb-name>
  .....
  <prim-key-class>p.PkB</prim-key-class>
  .....
  <cmp-field>
    <field-name>id1B</field-name>
  </cmp-field>
  <cmp-field>
    <field-name>id2B</field-name>
  </cmp-field>
  .....
</entity>
.....
<relationships>
  <ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
    <ejb-relationship-role>
      <!-- A => B -->
            <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>A</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>b</cmr-field-name>
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <!-- B => A -->
      <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>B</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>a</cmr-field-name>
      </cmr-field>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
.....
```

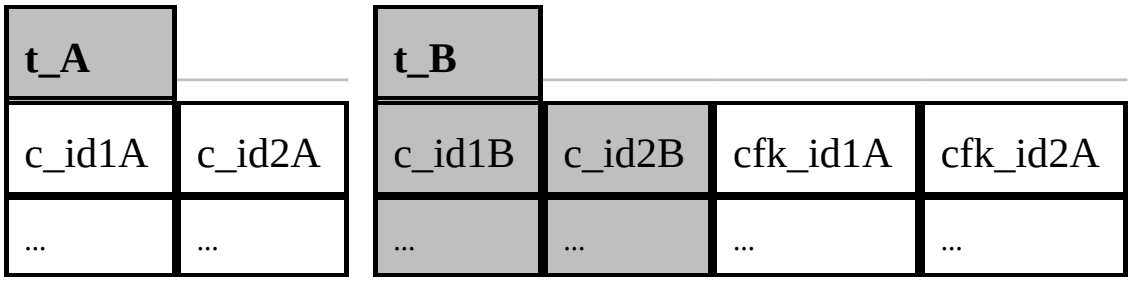
Database mapping

Two mappings are possible, one or another of the tables may hold the foreign key.

Case 1:



Case 2:



JOnAS Deployment Descriptor

Case 1:

```
.....
<jonas-ejb-relation>
  <ejb-relation-name>a-b</ejb-relation-name>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_id1b</foreign-key-jdbc-name>
      <key-jdbc-name>c_id1b</key-jdbc-name>
    </foreign-key-jdbc-mapping>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_id2b</foreign-key-jdbc-name>
      <key-jdbc-name>c_id2b</key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
</jonas-ejb-relation>
.....
```

Case 2:

```
.....
<jonas-ejb-relation>
  <ejb-relation-name>a-b</ejb-relation-name>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_id1a</foreign-key-jdbc-name>
      <key-jdbc-name>c_id1a</key-jdbc-name>
    </foreign-key-jdbc-mapping>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_id2a</foreign-key-jdbc-name>
      <key-jdbc-name>c_id2a</key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
</jonas-ejb-relation>
.....
```

For the default mapping (values), the foreign key is in the table of the source bean of the first ejb−relationship−role of the ejb−relation. In the example, the default mapping corresponds to case 1, since the ejb−relationship−role a2b is the first defined in the ejb−relation a−b.

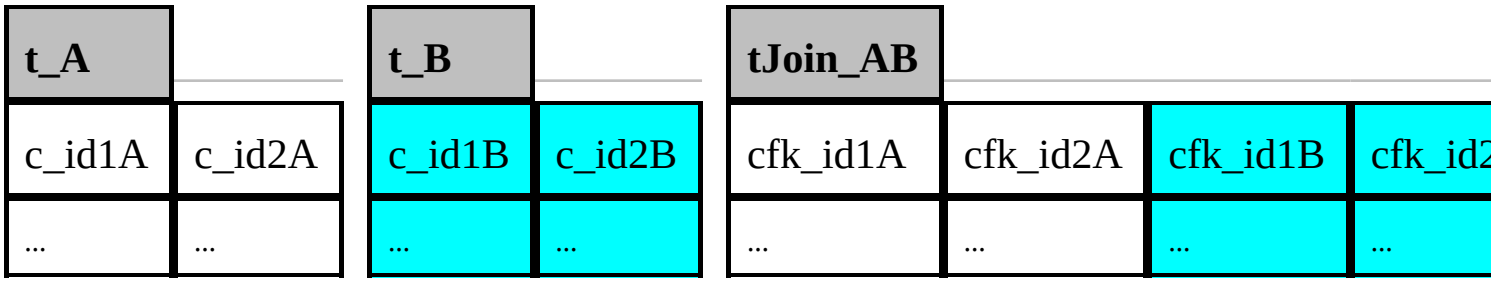
N−M unidirectional relationships

Standard Deployment Descriptor

```
.....
<entity>
  <ejb-name>A</ejb-name>
  .....
  <cmp-field>
    <field-name>id1A</field-name>
  </cmp-field>
  <cmp-field>
    <field-name>id2A</field-name>
  </cmp-field>
  .....
</entity>
.....
<entity>
  <ejb-name>B</ejb-name>
  .....
  <cmp-field>
    <field-name>id1B</field-name>
  </cmp-field>
  <cmp-field>
    <field-name>id2B</field-name>
  </cmp-field>
  .....
</entity>
.....
```

```
<relationships>
  <ejb-relation>
    <ejb-relation-name>a-b</ejb-relation-name>
    <ejb-relationship-role>
      <!-- A => B -->
            <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>A</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>b</cmr-field-name>
        <cmr-field-type>java.util.Collection</cmr-field-type>
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <!-- B => A -->
      <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>B</ejb-name>
      </relationship-role-source>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
.....
```

Database mapping



In this case, there is a join table composed of the foreign keys of each entity bean table.

JOnAS Deployment Descriptor

```
.....
<jonas-ejb-relation>
  <ejb-relation-name>a-b</ejb-relation-name>
  <jdbc-table-name>tJoin_AB</jdbc-table-name>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>a2b</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_id1b</foreign-key-jdbc-name>
      <key-jdbc-name>c_id1b</key-jdbc-name>
    </foreign-key-jdbc-mapping>
```

```
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_id2b</foreign-key-jdbc-name>
      <key-jdbc-name>c_id2b</key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
  <jonas-ejb-relationship-role>
    <ejb-relationship-role-name>b2a</ejb-relationship-role-name>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_id1a</foreign-key-jdbc-name>
      <key-jdbc-name>c_id1a</key-jdbc-name>
    </foreign-key-jdbc-mapping>
    <foreign-key-jdbc-mapping>
      <foreign-key-jdbc-name>cfk_id2a</foreign-key-jdbc-name>
      <key-jdbc-name>c_id2a</key-jdbc-name>
    </foreign-key-jdbc-mapping>
  </jonas-ejb-relationship-role>
</jonas-ejb-relation>
.....
```

Guide du programmeur EJB : Développer des Beans "Message−driven"

Contenu de ce guide :

La spécification EJB 2.0 défini un nouveau type de composant EJB permettant la réception de messages. Il permet d'invoquer des méthodes de l'EJB de manière asynchrone. Un Bean Message−driven (appelé aussi MDB pour Message Driven Bean) est un EJB qui n'est ni un Bean Entité, ni un Bean Session, et qui joue le rôle de "MessageListener" pour les messages JMS.

La spécification EJB 2.0 contient les informations détaillées sur les MDB. La spécification JMS (Java Message Service) 1.0.2 contient les informations détaillées sur JMS. Ce chapitre traite de l'utilisation des Beans Message−driven dans l'environnement JOnAS.

Description d'un Bean Message−driven

Un Bean Message−driven est un EJB qui se comporte comme un "MessageListener" JMS, c'est−à−dire qui reçoit des messages JMS et les traite de manière asynchrone ; il implémente la méthode *onMessage(javax.jms.Message)* définie dans l'interface *javax.jms.MessageListener*. Il est associé à une "destination JMS", c'est−à−dire une "Queue" pour les communications "point à point" ou à un "Topic" pour les communications "publish/suscribe" (publication/souscription). La méthode *onMessage* est activée à la réception d'un message envoyé par un client JMS associé à la destination. Il est également possible de configurer des filtres de message JMS afin de sélectionner les messages que le MDB doit recevoir.

Les messages JMS ne portent aucun contexte, et la méthode *onMessage* ne s'exécutera pas dans le contexte transactionnel pré−existant au moment de l'envoi. Cependant, une nouvelle transaction peut être initiée au moment de la réception (voir la section sur les "Aspects transactionnels"). La méthode *onMessage* peut appeller d'autres méthodes sur le Bean Message−driven lui−même ou sur d'autres Beans, et peut impliquer d'autres ressources par exemple en accédant à une base de donnée ou en envoyant des messages. Ces ressources sont accédées de la même manière que pour les autres EJB, c'est−à−dire grâce à une référence déclarée dans le descripteur de déploiement.

Le conteneur EJB de JOnAS maintient un pool d'instances de MDB, permettant le traitement d'importants volumes de messages. Un Bean Message−driven est similaire sur de nombreux points à un Bean Session sans état : ses instances ont une vie courte, il ne conserve aucun état d'un client spécifique, et plusieurs instances peuvent être exécutées en même temps.

Développer un Bean Message−driven

La classe du MDB doit implémenter les interfaces *javax.jms.MessageListener* et *javax.ejb.MessageDrivenBean*. En plus de la méthode *onMessage*, les méthodes suivantes doivent être implémentées :

- Un constructeur publique sans arguments.
- *public void ejbCreate()*: sans arguments, appelée au moment de l'instanciation. Elle peut être utilisée pour obtenir des ressources au moment de la création, comme des connexions à des backends spécifiques (base de données, ...).
- *public void ejbRemove()*: utilisée pour libérer les ressources obtenues au moment de la création.
- *public void setMessageDrivenContext(MessageDrivenContext mdc)*: appelée par le conteneur aprés la création de l'instance, et sans contexte transactionnel. JOnAS fournit au Bean un contexte qui peut être utilisé pour la gestion des transactions, ex : *setRollbackOnly()*, *getRollbackOnly()*, *getUserTransaction()*.

Exemple d'un Bean Message−driven :

```
public class MdbBean  implements MessageDrivenBean, MessageListener {

    private transient MessageDrivenContext mdbContext;

    public MdbBean() {}

    public void setMessageDrivenContext(MessageDrivenContext ctx) {
        mdbContext = ctx;
    }

    public void ejbRemove() {}

    public void ejbCreate() {}

    public void onMessage(Message message) {
        try {
            TextMessage mess = (TextMessage)message;
            System.out.println( "Message reçu: "+mess.getText());
        }catch(JMSException ex){
            System.err.println("Exception : "+ex);
        }
    }
}
```

La destination associée au Bean Message−driven est spécifiée dans le descripteur de déploiement du Bean. Une destination est un objet JMS, accessible par JNDI. La description d'un MDB dans le descripteur de déploiement EJB 2.0 contient les éléments suivants, qui sont spécifiques aux MDB :

- le mode d'acquittement JMS : auto−acknowledge ou dups−ok−acknowledge (référez vous à la spécification JMS pour une description des différents modes d'acquittement).
- éventuellement un sélecteur de messages : c'est un concept de JMS qui permet de filtrer les messages envoyés.

- un tag message−driven−destination, contenant le type de la destination (Queue ou Topic) et la durabilité de la souscription (dans le cas d'un Topic).

Exemple d'un descripteur de déploiement pour MDB :

```
<enterprise−beans>
  <message−driven>
    <description>Describe here the message driven bean Mdb</description>
    <display−name>Message Driven Bean Mdb</display−name>
    <ejb−name>Mdb</ejb−name>
    <ejb−class>samplemdb.MdbBean</ejb−class>
    <transaction−type>Container</transaction−type>
    <message−selector>Weight >= 60.00 AND LName LIKE 'Sm_th'</message−selector>
    <message−driven−destination>
      <destination−type>javax.jms.Topic</destination−type>
      <subscription−durability>NonDurable</subscription−durability>
    </message−driven−destination>
    <acknowledge−mode>Auto−acknowledge</acknowledge−mode>
  </message−driven>
</enterprise−beans>
```

Si le type de transaction est "container", le comportement transactionnel des méthodes du MDB est défini comme pour les autres types d'EJB dans le descripteur de déploiement, comme dans l'exemple suivant :

```
<assembly−descriptor>
  <container−transaction>
    <method>
      <ejb−name>Mdb</ejb−name>
      <method−name>*</method−name>
    </method>
    <trans−attribute>Required</trans−attribute>
  </container−transaction>
</assembly−descriptor>
```

Pour la méthode *onMessage* seuls les types *Required* et *NotSupported* peuvent être utilisés, étant donné qu'il n'existe pas de contexte transactionnel pré−existant.

Pour le sélecteur de messages spécifié dans l'exemple précédent, les messages JMS envoyés doivent avoir 2 propriétés : "Weight" et "LName," par exemple assignées par le programme client qui envoie les messages JMS :

```
        message.setDoubleProperty("Weight",75.5);
        message.setStringProperty("LName","Smith");
```

Ainsi ce message sera reçu par le MDB. La syntaxe des filtres de messages est basée sur un sous−ensemble de SQL92. Seuls les messages dont l'en−tête et les propriétés correspondent au filtre sont délivrés. Référez vous à la spécification JMS pour plus de détails.

Le nom JNDI de la destination associée avec le MDB est défini dans le descripteur de déploiement spécifique de JOnAS, grâce à un élément *jonas−message−driven* :

```
<jonas−message−driven>
  <ejb−name>Mdb</ejb−name>
  <jonas−message−driven−destination>
    <jndi−name>sampleTopic</jndi−name>
  </jonas−message−driven−destination>
</jonas−message−driven>
```

Ensuite, l'application cliente peut envoyer des messages au MDB par l'objet Destination obtenu par JNDI comme dans l'exemple suivant :

```
Queue q = context.lookup("sampleTopic");
```

Si le client envoyant le message est lui même un composant EJB, il est préférable qu'il utilise une référence sur la ressource dans l'environnement EJB pour obtenir l'objet Destination. L'utilisation de ces références est décrite dans le Guide JMS.

Aspects administration

JOnAS utilise une implémentation existante de JMS, ex : JORAM, SwiftMQ.

Par défaut, le développeur de MDB et le déployeur ne sont pas concernés par l'administration JMS. Cela signifie que le développeur/déployeur ne crée pas de "Connection factories" ou de destination JMS (ce qui est nécessaire pour effectuer des opérations JMS depuis un composant EJB); ils définissent simplement le type de la destination dans le descripteur de déploiement et identifient son nom JNDI dans le descripteur de déploiement spécifique de JOnAS, comme décrit dans la section précédente. Cela signifie que JOnAS va implicitement créer les objets nécessaires en utilisant l'API d'administration propriétaire de l'implémentation JMS utilisée (l'API d'administration n'a pas encore été standardisée). Pour effectuer ces opérations d'administration, JOnAS utilise un "wrapper" vers l'API d'administration du provider JMS. Pour Joram, le wrapper est *org.objectweb.jonas_jms.JmsAdminForJoram* (qui est le wrapper par défaut défini pour la propriété *jonas.service.jms.mom* dans le fichier ***jonas.properties***). Pour SwiftMQ, la classe *com.swiftmq.appserver.jonas.JmsAdminForSwiftMQ* peut être obtenue sur le site de SwiftMQ.

Le déployeur doit ajouter le service 'jms' dans la liste des services JOnAS. Pour l'exemple fourni, le fichier jonas.properties doit contenir :

```
jonas.services                  registry,security,jtm,dbm,jms,ejb      // The jms service must be
jonas.service.ejb.descriptors   samplemdb.jar
jonas.service.jms.topics        sampleTopic        // not mandatory
```

L'objet Destination peut exister, ou pas. Le serveur EJB ne créera pas la destination JMS si elle existe déja. Le topic `sampleTopic` doit être déclaré explicitement, seulement si JOnAS doit le créer en premier, avant que le MDB ne soit chargé ou s'il est utilisé par un autre client avant que le MDB ne soit chargé. En général il n'est pas nécessaire de

le déclarer.

JOnAS utilise un ***pool de threads*** pour exécuter les instances de Beans Messages à la reception de messages, permettant ainsi à de large volumes de messages d'être traités simultanement. Comme expliqué précedemment, les instances de MDB sont sans état et plusieurs instances peuvent être exécutées simultanément. La taille par défaut du pool de thread est de 10, et peut être adaptée par la propriété ***jonas.service.ejb.mdbthreadpoolsize***, spécifiée dans le fichier ***jonas.properties*** :

```
jonas.service.ejb.mdbthreadpoolsize   50
```

Exécuter un Bean Message−driven

Pour déployer et exécuter un Bean Message−driven, les étapes suivantes doivent être réalisées :

- Vérifier que le "registry" JNDI est démarré.
- Démarrer le MOM (Message−Oriented Middleware), c'est−à−dire votre implémentation de JMS. Référez vous à la rubrique "Démarrer le MOM (Message−Oriented Middleware) ."
- Créer et enregistrer la destination JMS qui sera utilisée par le Bean Message−driven dans le JNDI..

  Cela peut être fait automatiquement par le service JMS ou explicitement par les moyens d'administration propriétaires du provider JMS (Administration JMS). Le service JMS crée l'objet Destination si celui ci est déclaré dans le fichier ***jonas.properties*** (comme spécifié dans la partie précedente).
- Déployer le Bean Message−driven dans JOnAS.

  Notez que, si l'objet Destination n'est pas encore créé, le conteneur EJB va demander au service JMS de le créer à partir des informations contenues dans le descripteur de déploiement.
- Lancer l'application cliente de l'EJB.

  Lorsque vous utilisez JMS, il est trés important de stopper JOnAS en utilisant la commande ***jonas stop***; il ne doit pas être stoppé directement en tuant le processus.

Démarrer le MOM (Message−Oriented Middleware)

Si la propriété ***jonas.services*** contient le service jms, alors le service JMS de JONAS sera lancé et essaiera de démarrer l'implémentation JMS (le MOM).

Il y a trois possibilités pour démarrer le MOM :

1. ***Démarrer le MOM dans la même JVM que JOnAS***

   C'est le comportement par défaut qui est obtenu en mettant à `true` la propriété ***jonas.service.jms.collocated*** dans le fichier ***jonas.properties***.

   ```
   jonas.services               security,jtm,dbm,jms,ejb  // The jms service must be in the l
   ```

```
jonas.service.jms.collocated  true
```

Dans ce cas, le MOM est automatiquement lancé par le service JMS de JOnAS au démarrage (commande `jonas start`).

2. ***Démarrer le MOM dans une JVM séparée***

   Le MOM Joram peut être lancé par la commande :

   ```
   JmsServer
   ```

   Pour les autres MOMs, la commande propriétaire doit être utilisée.

   La propriété ***jonas.service.jms.collocated*** doit être à `false` dans le fichier ***jonas.properties***. Définir cette propriété est suffisant si la JVM de JORAM tourne sur la même machine que JONAS, et si le MOM est démarré avec les options par défaut (fichier de configuration ***a3servers.xml*** non modifié dans JONAS_BASE/conf ou JONAS_ROOT/conf si JONAS_BASE n'est pas défini).

   ```
   jonas.services                security,jtm,dbm,jms,ejb   // The jms service must be in the
   jonas.service.jms.collocated  false
   ```

   Pour utiliser une configuration spécifique du MOM, comme changer l'host par défaut (localhost) ou le port par defaut (16010), vous devez ajouter la propriété supplémentaire ***jonas.service.jms.url***.

3. ***Démarrer le MOM sur une autre machine***

   Cela nécessite de définir la propriété ***jonas.service.jms.url***. Lorsque vous utilisez JORAM, cette valeur doit être l'URL de JORAM `joram://host:port` où `host` est le nom de la machine, et `port` est le port de connexion (par défaut, 16010). Pour SwiftMQ, la valeur doit ressembler à `smqp://host:4001/timeout=10000`.

   ```
   jonas.services                security,jtm,dbm,jms,ejb   // The jms service must be in the
   jonas.service.jms.collocated  false
   jonas.service.jms.url         joram://host2:16010
   ```

- ***Modifier la configuration par défaut de JORAM***

  L'url de connexion par défaut peut avoir besoin d'être changée. Cela nécessite de modifier le fichier de configuration ***a3servers.xml*** fourni dans JOnAS dans le répertoire JONAS_ROOT/conf. JOnAS doit être configuré avec la propriété ***jonas.service.jms.collocated*** à `false`, et la propriété ***jonas.service.jms.url*** à `joram://host:port`. De plus le MOM doit être démarré avec la commande JmsServer. Cette commande définit une propriété `Transaction` à `fr.dyade.aaa.util.NullTransaction`. Si les messages ont besoin d'être persistants, changez l'option `−DTransaction=fr.dyade.aaa.util.NullTransaction` en `−DTransaction=fr.dyade.aaa.util.ATransaction`. Référez vous à la documentation de

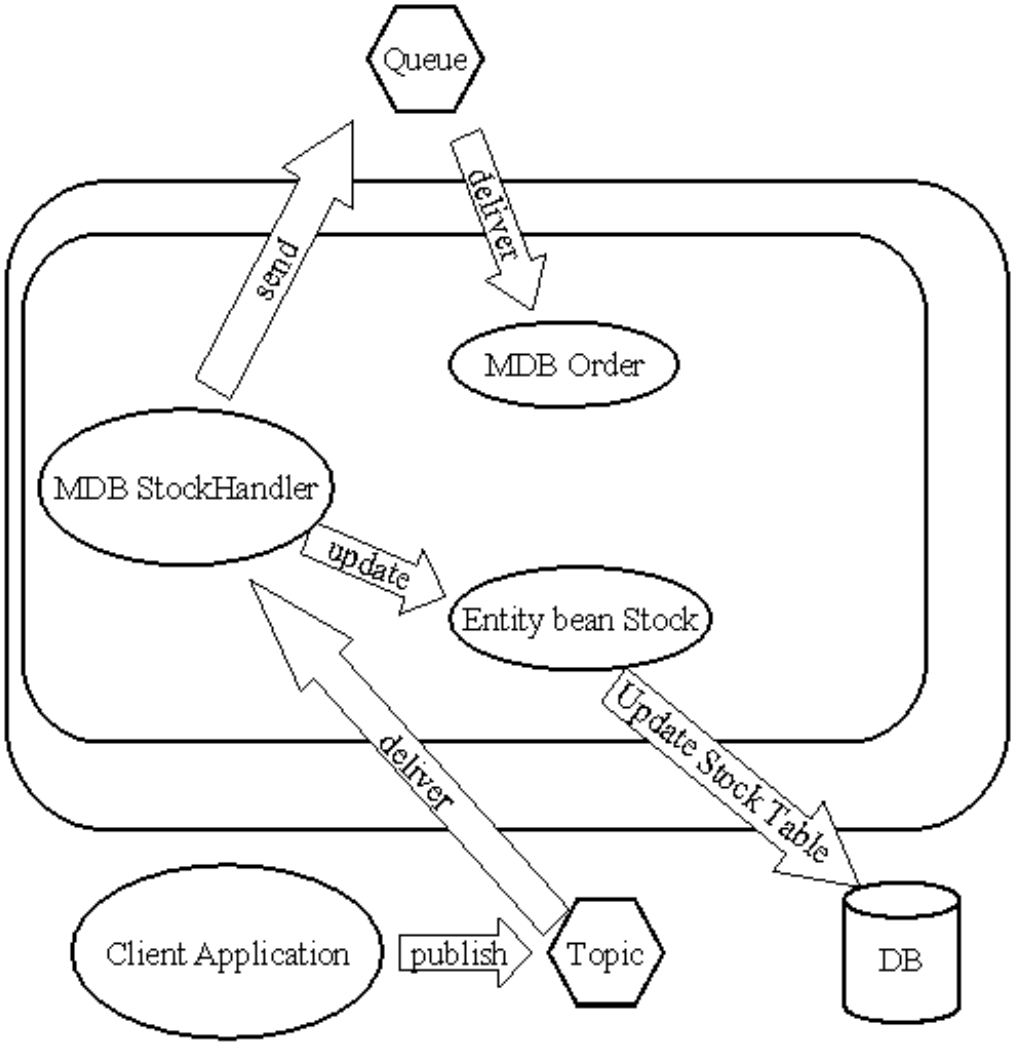
JORAM pour plus de détails à propos de cette commande. Pour définir une configuration plus complexe (e.g., distribution, multi−servers), référez vous à la documentation de JORAM http://joram.objectweb.org.

Aspects Transactionnels

Comme le contexte transactionnel ne peut pas être transmis avec le message (selon la spécification EJB 2.0), un Bean Message−driven ne s'éxécutera jamais dans une transaction existante. Cependant, une transaction peut être démarrée à l'exécution de la méthode onMessage soit en spécifiant l'attribut de transaction "required" (transactions gérées par le conteneur) ou en démarrant explicitement une transaction (transactions gérées par le Bean). Dans le second cas, la réception du message ne fera pas partie de la transaction. Dans le premier cas, le conteneur va démarrer la transaction avant de retirer le message de la queue. Si la méthode onMessage invoque d'autre EJBs, le conteneur va passer le contexte transactionnel avec l'appel. Ainsi la transaction démarrée dans onMessage peut impliquer plusieurs opérations, comme accéder à une base de données (via un Bean Entité ou en utilisant une "DataSource"), ou envoyer des messages.

Exemple

JOnAS fournit des exemples dans le répertoire ***examples/src/mdb***.
samplemdb est un exemple trés simple dont le code illustre comment utiliser des Beans Message−driven.
sampleappli est un exemple plus complexe qui montre comment l'envoi de message et la mise à jour d'une base données via JDBC peuvent se faire dans la même transaction distribuée.
La figure suivante illustre l'architecture de l'application.


Queue
send
deliver
MDB Order
MDB StockHandler
update
Entity bean Stock
Update Stock Table
deliver
Client Application
publish
Topic
DB

Cet exemple comporte 2 Beans Message−driven :

- ***StockHandlerBean*** est un Bean Message qui écoute un Topic et reçoit des messages. La méthode onMessage s'éxecute dans le contexte transactionnel démarré par le conteneur. Il envoi un message texte sur une Queue (OrderQueue) et met à jour l'élément Stock en décrémentant la quantité du stock. Si la quantité de stock devient négative, une exception est reçue et la transaction est marquée "rollback".
- ***OrderBean*** est un autre Bean Message−driven écoutant la queue OrderQueue. A la réception d'un message Text, il écrit la chaine correspondante dans une nouvelle ligne du fichier "Order.txt".

L'exemple inclut également un Bean Entité CMP ***Stock*** qui représente les données de la table Stock.

Un élément de Stock est composé d'un Stockid (String), qui est la clé primaire, ainsi que de Quantity (int). La méthode decreaseQuantity(int qty) décrémente la quantité pour le stockid correspondant, mais peut déclencher une RemoteException "Negative stock."

L'application client ***SampleAppliClient*** est un client qui envoie plusieurs messages sur le topic *StockHandlerTopic*. Il utilise des messages "Map" avec 3 champs: "CustomerId," "ProductId," "Quantity." Avant d'envoyer les messages, le client appel ***EnvBean*** afin de créer la table *StockTable* dans la base de donnée avec les valeurs connues afin de tester le résultat des mises à jour à la fin du test. 11 messages sont envoyés, les transactions correspondantes sont validées et le dernier message envoyé cause le rollback de la transaction.

Compiler l'exemple

Pour compiler `examples/src/mdb/sampleappli`, utilisez *Ant* avec le fichier `$JONAS_ROOT/examples/src/build.xml`.

Exécuter l'exemple

La configuration par défaut du service JMS dans le fichier `jonas.properties` est le suivant:

```
jonas.services                 jmx,security,jtm,dbm,jms,ejb           // The jms service must be
jonas.service.ejb.descriptors  sampleappli.jar
jonas.service.jms.topics       StockHandlerTopic
jonas.service.jms.queues       OrdersQueue
jonas.service.jms.collocated   true
```

Cela indique que le serveur JMS sera lancé dans la même JVM que JOnAS et que les objets JMS `StockHandlerTopic` (Topic) et `OrdersQueue` (Queue) seront créés et enregistrés dans le JNDI, s'ils n'existent pas déjà.

- Démarrer le serveur JOnAS.

  ```
  jonas start
  ```
- Déployer sampleappli.

```
jonas admin −a sampleappli.jar
```

- Lancer le client.

```
jclient sampleappli.SampleAppliClient
```

- Stopper le serveur.

```
jonas stop
```

Régler le pool de Beans Messages

JOnAS maintient un pool pour chaque Bean Message−driven. Le pool peut être configuré dans le descripteur de déploiement spécifique de JOnAS grâce aux tags suivants :

min−pool−size

Cet entier optionnel représente le nombre minimum d'instances qui seront créées lorsque le Bean est chargé. Cela va améliorer le temps de création des instances, au moins pour les premières. La valeur par défaut est 0.

max−cache−size

Cet entier optionnel représente le nombre maximum d'instances de *ServerSession* qui peuvent être créées en mémoire. L'objectif de cette valeur est de permettre la montée en charge de JOnAS. La politique est la suivante :
Quand le *ConnectionConsumer* demande une instance de *ServerSession* (afin de délivrer un message) JOnAS essaie de prendre une instance dans *ServerSessionPool*. Si le pool est vide, une nouvelle instance est créée, seulement si le nombre d'instances déjà créées est plus petit que le paramètre max−cache−size. Quand max−cache−size est atteint, *ConnectionConsumer* est bloqué et ne peut plus délivrer de messages jusqu'à ce qu'un *ServerSession* retourne au pool. Un *ServerSession* est remis dans le pool à la fin de la méthode *onMessage*.
La valeur par défaut est "pas de limite" (ce qui signifie qu'une nouvelle instance de *ServerSession* est toujours créée lorsque le pool est vide).

exemple

```
<jonas−ejb−jar>
  <jonas−message−driven>
    <ejb−name>Mdb</ejb−name>
    <jndi−name>mdbTopic</jndi−name>
    <max−cache−size>20</max−cache−size>
    <min−pool−size>10</min−pool−size>
  </jonas−message−driven>
</jonas−ejb−jar>
```

Guide du programmeur EJB: Comportement transactionnel

Public visé et contenu

Ce guide est destiné au fournisseur d' EJB, c'est−à−dire à la personne chargée de développer des composants logiciels coté serveur. Le guide explique comment définir le comportement transactionnel d'une application à base d'EJBs.

Contenu du guide :

Gérer les transactions de manière déclarative

Pour les transactions gérées par le conteneur, le comportement transactionnel d'un EJB est défini au moment de la configuration et fait partie de l'élement "assembly−descriptor" du descripteur de déploiement EJB standard. Il est possible de déclarer un comportement transactionnel commun pour toutes les méthodes du Bean, ou de définir le comportement transactionnel au niveau de chaque méthode. Ceci est fait en spécifiant un attribut transactionnel, qui peut prendre une des valeurs suivantes :

- ***NotSupported:*** si la méthode est appelée dans une transaction, cette transaction est suspendue durant l'exécution de la méthode.
- ***Required***: si la méthode est appelée dans une transaction, la méthode est exécutée dans cette transaction, sinon, une nouvelle transaction est démarrée pour l'éxecution de cette méthode et est committée avant que le résultat ne soit envoyé au client.
- ***RequiresNew***: la méthode sera toujours exécutée dans une nouvelle transaction. Cette transaction sera démarrée pour l'exécution de la méthode et sera committée avant que le résultat ne soit envoyé au client. Si la méthode est appellée dans une transaction, celle−ci est suspendue avant que la nouvelle ne soit créée, et reprise lorsque là nouvelle est terminée.
- ***Mandatory***: la méthode doit toujours être appelée dans une transaction. Dans le cas contraire, le conteneur déclenche une exception *TransactionRequired*.
- ***Supports***: la méthode est appelée dans le contexte transactionnel du client ; si celui−ci n'a pas de transaction associée, la méthode est appelée sans contexte transactionnel.
- ***Never***: Le client doit appeler cette méthode sans contexte transactionnel ; dans le cas contraire le conteneur déclenche une exception *java.rmi.RemoteException*.

Ceci est illustré dans la tableau suivant :

Attribut transactionel	Transaction du client	Transaction associée à l'appel de la méthode de l'EJB

NotSupported	– T1	– –
Required	– T1	T2 T1
RequiresNew	– T1	T2 T2
Mandatory	– T1	erreur T1
Supports	– T1	– T1
Never	– T1	– erreur

Dans le descripteur de déploiement, les attributs transactionnels sont définis dans l'élément "assembly−descriptor" comme cela :

```
<assembly−descriptor>
  <container−transaction>
    <method>
    <ejb−name>AccountImpl</ejb−name>
    <method−name>*</method−name>
    </method>
    <trans−attribute>Supports</trans−attribute>
  </container−transaction>
  <container−transaction>
    <method>
    <ejb−name>AccountImpl</ejb−name>
    <method−name>getBalance</method−name>
    </method>
    <trans−attribute>Required</trans−attribute>
  </container−transaction>
  <container−transaction>
    <method>
    <ejb−name>AccountImpl</ejb−name>
    <method−name>setBalance</method−name>
    </method>
    <trans−attribute>Mandatory</trans−attribute>
  </container−transaction>
</assembly−descriptor>
```

Dans cet exemple, pour toutes les méthodes du Bean AccountImpl qui ne spécifient pas explicitement d'attribut transactionnel, l'attribut par défaut est Support (défini au niveau du Bean), et les attributs transactionnels des méthodes getBalance et setBalance sont respectivement définis à Required et Mandatory.

Transactions gérées par le Bean

Un Bean qui gère ses transactions lui−même doit définir le champ `transaction−type` dans le descripteur de déploiement standard comme suit :

```
  <transaction−type>Bean</transaction−type>
```

Afin de démarquer son contexte transactionel dans un Bean gérant les transactions lui−même, le programmeur doit utiliser l'interface *javax.transaction.UserTransaction*, qui est définie comme un objet géré du serveur EJB et qui peut être obtenu en utilisant la méthode *EJBContext.getUserTransaction()* (sur l'objet SessionContext ou EntityContext selon que la méthode fait partie d'un EJB Session ou Entité). L'exemple suivant montre l'utilisation des transactions dans un Bean Session gérant son contexte transactionnel lui−même; l'objet UserTransaction est obtenu à partir de l'objet sessionContext, qui a été initialisé dans la méthode setSessionContext (réferez vous à à l'exemple d'un Bean Session).

```
public void doTxJob() throws  RemoteException {
     UserTransaction ut = sessionContext.getUserTransaction();
     ut.begin();
     ... // opérations transactionnelles
     ut.commit();
}
```

Une autre manière d'obtenir l'objet UserTransaction est de le chercher dans le JNDI sous le nom `java:comp/UserTransaction`.

Gestion des transactions distribuées

Comme expliqué dans la section précédente, le comportement transactionnel peut être déclaratif ou être codé dans le client lui−même. Dans tous les cas, l'aspect distribué d'une transaction est totalement transparent au programmeur et au déployeur de l'application. Ceci signifie qu'une transaction peut impliquer plusieurs Beans distribués sur plusieurs serveurs EJB et que la plate−forme gère elle−même la transaction globale. Cela est fait par l'application du protocole de validation à deux phases ("two−phase commit") entre les différents serveurs, et le programmeur n'a rien à faire.

Une fois que les Beans ont été développés et que l'application a été assemblée, il est possible pour le déployeur ou pour l'administrateur de configurer la distribution des différents Beans sur plusieurs machines et entre un ou plusieurs serveurs JOnAS. Ceci peut être fait sans impact sur le code du Bean ou le descripteurs de déploiement. La configuration distribuée est spécifiée au moment du démarrage du serveur. Les informations suivantes doivent être données :

- quels Beans doivent être gérés par le serveur JOnAS.

- si le moniteur de transaction (Java Transaction Monitor) est dans la même machine virtuelle ou non.

Pour ceci, vous devez définir deux propriétés dans le fichier *jonas.properties* : *jonas.service.ejb.descriptors* et *jonas.service.jtm.remote*. La première liste les EJBs qui doivent être gérés par le serveur (en spécifiant le nom de leur fichier ejb−jar) et la seconde définit le mode de lancement du Java Transaction Monitor (JTM) :

- si la valeur est `true`, le JTM est distant, c'est à dire que le JTM doit être démarré avant dans une autre JVM.
- si la valeur est `false`, le JTM est local, c'est à dire qu'il sera lancé dans la même JVM que le serveur EJB.

Exemple:

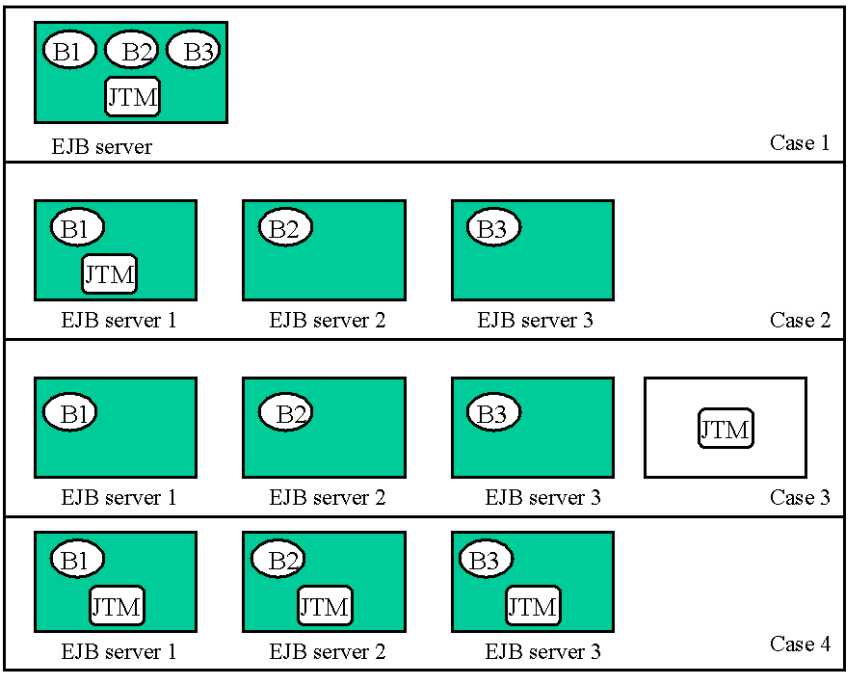
```
jonas.service.ejb.descriptors       Bean1.jar, Bean2.jar
jonas.service.jtm.remote            false
```

Le Java Transaction Monitor peut être démarré en mode stand−alone par la commande :

```
TMServer
```

En utilisant ces configurations, il est possible d'adapter la distribution des EJBs en fonction des ressources (cpu et données) afin d'optimiser les performances.

La figure suivante illustre 4 cas de configurations pour 3 Beans :.



1. Cas 1: Les 3 Beans B1, B2, et B3 sont dans le même serveur JOnAS qui inclut le Java Transaction Monitor.

2. Cas 2: Les 3 Beans sont localisés dans différents serveurs JOnAS, dont l'un d'entre eux exécute le Java Transaction Monitor, gérant les transactions globales.
3. Cas 3: Les 3 Beans sont localisés dans différents serveurs JOnAS et le Java Transaction Monitor s'exécute en mode stand−alone.
4. Cas 4: Les 3 Beans sont localisés dans différents serveurs JOnAS. Chaque serveur exécute un Java Transaction Monitor. Un des JTM a le rôle de moniteur maitre, et les 2 autres ont le rôle d'esclaves.

Ces différentes configurations peuvent être obtenues en démarrant les serveurs JOnAS et éventullement le JTM (cas 3) avec les propriétés adéquates. La meilleure solution dépend du type de l'application :

- si les Beans doivent s'exécuter sur la même machine, avec la même configuration du serveur, le cas 1 est plus approprié ;
- si les Beans doivent s'exécuter sur différentes machines, le cas 4 est plus approprié, car il favorise la gestion locale des transactions ;
- si les Beans doivent s'exécuter sur la même machine, mais ont besoin de configurations du serveur différentes, le cas 2 est plus approprié.

EJB Programmer's Guide: Enterprise Bean Environment

Target Audience and Content

The target audience for this guide is the Enterprise Bean provider, i.e. the person in charge of developing the software components on the server side. It describes how an enterprise component can refer to values, resources, or other components in a way that is configurable at deployment time.

The content of this guide is the following:

Introduction

The enterprise bean environment is a mechanism that allows customization of the enterprise bean's business logic during assembly or deployment. The environment is a way for a bean to refer to a value, to a resource, or to another component so that the code will be independent of the actual referred object. The actual value of such environment references (or variables) is set at deployment time, according to what is contained in the deployment descriptor. The enterprise bean's environment allows the enterprise bean to be customized without the need to access or change the enterprise bean's source code.

The enterprise bean environment is provided by the container (i.e. the JOnAS server) to the bean through the JNDI interface as a JNDI context. The bean code accesses the environment using JNDI with names starting with "java:comp/env/".

Environment Entries

The bean provider declares all the bean environment entries in the deployment descriptor via the *env−entry* element. The deployer can set or modify the values of the environment entries.

A bean accesses its environment entries with a code similar to the following:

```
InitialContext ictx = new InitialContext();
Context myenv = ictx.lookup("java:comp/env");
Integer min = (Integer) myenv.lookup("minvalue");
Integer max = (Integer) myenv.lookup("maxvalue");
```

In the standard deployment descriptor, the declaration of these variables are as follows:

```
<env−entry>
  <env−entry−name>minvalue</env−entry−name>
  <env−entry−type>java.lang.Integer</env−entry−type>
  <env−entry−value>12</env−entry−value>
</env−entry>
<env−entry>
  <env−entry−name>maxvalue</env−entry−name>
  <env−entry−type>java.lang.Integer</env−entry−type>
  <env−entry−value>120</env−entry−value>
</env−entry>
```

Resource References

The resource references are another examples of environment entries. For such entries, using subcontexts is recommended:

- *java:comp/env/jdbc* for references to DataSources objects.
- *java:comp/env/jms* for references to JMS connection factories.

In the standard deployment descriptor, the declaration of a resource reference to a JDBC connection factory is:

```
<resource−ref>
  <res−ref−name>jdbc/AccountExplDs</res−ref−name>
  <res−type>javax.sql.DataSource</res−type>
  <res−auth>Container</res−auth>
</resource−ref>
```

And the bean accesses the datasource as in the following:

```
InitialContext ictx = new InitialContext();
DataSource ds = ictx.lookup("java:comp/env/jdbc/AccountExplDs");
```

Binding of the resource references to the actual resource manager connection factories that are configured in the EJB server is done in the JOnAS−specific deployment descriptor using the *jonas−resource* element.

```
<jonas−resource>
  <res−ref−name>jdbc/AccountExplDs</res−ref−name>
  <jndi−name>jdbc_1</jndi−name>
</jonas−resource>
```

Resource Environment References

The resource environment references are another example of environment entries. They allow the Bean Provider to refer to administered objects that are associated with resources (for example, JMS Destinations), by using *logical* names. Resource environment references are defined in the standard deployment descriptor.

```
<resource-env-ref>
  <resource-env-ref-name>jms/stockQueue</resource-env-ref-name>
  <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
</resource-env-ref>
```

Binding of the resource environment references to administered objects in the target operational environment is done in the JOnAS-specific deployment descriptor using the *jonas-resource-env* element.

```
<jonas-resource-env>
  <resource-env-ref-name>jms/stockQueue</resource-env-ref-name>
  <jndi-name>myQueue<jndi-name>
</jonas-resource-env>
```

EJB References

The EJB reference is another special entry in the enterprise bean's environment. EJB references allow the Bean Provider to refer to the homes of other enterprise beans using *logical* names. For such entries, using the subcontext *java:comp/env/ejb* is recommended.

The declaration of an EJB reference used for accessing the bean through its **remote** home and component interfaces in the standard deployment descriptor is shown in the following example:

```
<ejb-ref>
  <ejb-ref-name>ejb/ses1</ejb-ref-name>
  <ejb-ref-type>session</ejb-ref-type>
  <home>tests.SS1Home</home>
  <remote>tests.SS1</remote>
</ejb-ref>
```

The declaration of an EJB reference used for accessing the bean through its **local** home and component interfaces in the standard deployment descriptor is shown in the following example:

```
<ejb-local-ref>
  <ejb-ref-name>ejb/locses1</ejb-ref-name>
  <ejb-ref-type>session</ejb-ref-type>
  <local-home>tests.LocalSS1Home</local-home>
  <local>tests.LocalSS1</local>
</ejb-local-ref>
```

If the referred bean is defined in the same ejb−jar or EAR file, the optional **ejb−link** element of the ejb−ref or ejb−local−ref element can be used to specify the actual referred bean. The value of the ejb−link element is the name of the target enterprise bean, i.e. the name defined in the ejb−name element of the target enterprise bean. If the target enterprise bean is in the same EAR file, but in a different ejb−jar file, the name of the ejb−link element should be the name of the target bean, prefixed by the name of the containing ejb−jar file followed by '#' (e.g. "My_EJBs.jar#bean1"). In the following example, the ejb−link element has been added to the ejb−ref (in the referring bean SSA) and a part of the description of the target bean (SS1) is shown:

```
<session>
  <ejb−name>SSA</ejb−name>
  ...
  <ejb−ref>
    <ejb−ref−name>ejb/ses1</ejb−ref−name>
    <ejb−ref−type>session</ejb−ref−type>
    <home>tests.SS1Home</home>
    <remote>tests.SS1</remote>
    <ejb−link>SS1</ejb−link>
  </ejb−ref>
...
</session>
...
<session>
  <ejb−name>SS1</ejb−name>
  <home>tests.SS1Home</home>
  <local−home>tests.LocalSS1Home</local−home>
  <remote>tests.SS1</remote>
  <local>tests.LocalSS1</local>
  <ejb−class>tests.SS1Bean</ejb−class>
  ...
</session>
 ...
```

If the bean SS1 was not in the same ejb−jar file as SSA, but in another file named product_ejbs.jar, the ejb−link element would have been:

```
    <ejb−link>product_ejbs.jar#SS1</ejb−link>
```

If the referring component and the referred bean are in separate files and not in the same EAR, the current JOnAS implementation does not allow use of the ejb−link element. In this case, to resolve the reference, the *jonas−ejb−ref* element in the JOnAS−specific deployment descriptor would be used to bind the environment JNDI name of the EJB reference to the actual JNDI name of the associated enterprise bean home. In the following example, it is assumed that the JNDI name of the SS1 bean home is SS1Home_one.

```
<jonas−session>
  <ejb−name>SSA</ejb−name>
  <jndi−name>SSAHome</jndi−name>
  <jonas−ejb−ref>
    <ejb−ref−name>ejb/ses1</ejb−ref−name>
```

```
    <jndi-name>SS1Home_one</jndi-name>
  </jonas-ejb-ref>
</jonas-session>
```

The bean locates the home interface of the other enterprise bean using the EJB reference with the following code:

```
InitialContext ictx = new InitialContext();
Context myenv = ictx.lookup("java:comp/env");
SS1Home home = (SS1Home)javax.rmi.PortableRemoteObject.narrow(myEnv.lookup("ejb/ses1"), SS1Home.c
```

Currently in JOnAS, it is not possible to explicitly assign a JNDI name for local home interfaces. This name is generated and is the JNDI name of the (remote) home followed by "_L" (this means that, even if there is only a local home on this bean, a jndi−name element must be defined).

Deprecated EJBContext.getEnvironment() method

JOnAS provides support for EJB 1.0−style definition of environment properties. EJB1.0 environment must be declared in the **ejb10−properties** sub−context. For example:

```
<env-entry>
  <env-entry-name>ejb10-properties/foo</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>foo value</env-entry-value>
</env-entry>
```

The bean can retrieve its environment with the following code:

```
SessionContext ctx;
Properties prop;
public void setSessionContext(SessionContext sc) {
    ctx = sc;
    prop = ctx.getEnvironment();
}
public mymethod() {
    String foo = prop.getProperty("foo");
    ...
}
```

Guide du programmeur EJB : Gestion de la sécurité

Public visé et contenu

Ce guide est destiné au fournisseur d'EJB, c'est−à−dire à la personne chargée de développer des composants logiciels coté serveur. Il explique comment doit être définie la sécurité d'une application à base d'EJB.

Contenu de ce guide :

Introduction

L'architecture EJB encourage le programmeur à éviter de coder directement la gestion de la securité dans l'implémentation des méthodes métier.

Gestion déclarative de la securité

L'assembleur de l'application J2EE peut définir une *vue de sécurité* des EJBs contenus dans le fichier ejb−jar.
La vue de sécurité consiste en un ensemble de *rôles de sécurité*. Un rôle de securité est une semantique regroupant un ensemble de permissions pour un type donné d'utilisateur de l'application. L'assembleur de l'application peut définir (de manière déclarative dans le descripteur de déploiement) des *permissions de méthode* pour chaque rôle de securité. Une permission de méthode permet d'invoquer un groupe spécifié de méthodes des interfaces Home et Remote d'un EJB. Les rôles de sécurité définis par l'assembleur de l'application représentent alors une vue simplifiée de la sécurité de l'application au déployeur; le déployeur ne voit qu'un petit ensemble de rôles de sécurité au lieu d'un grand nombre de permissions définies sur les méthodes.

Rôles de sécurité

L'assembleur de l'application peut définir un ou plusieurs *rôles de securité* dans le descripteur de déploiement. L'assembleur de l'application assigne alors un groupe de méthodes de l'EJB (interfaces Home ou Remote) à un rôle de sécurité, afin de sécuriser l'application.

La portée d'un rôle de sécurité défini dans l'élément `security−role` est au niveau de l'ejb−jar et inclut tous les EJB contenus dans cet ejb−jar.

```
...
<assembly−descriptor>
   <security−role>
      <role−name>tomcat</role−name>
   </security−role>
```

```
   ...
</assembly-descriptor>
```

Permissions de méthode

Aprés avoir défini les rôles de sécurité des EJB de l'ejb−jar, l'assembleur de l'application peut aussi spécifier les méthodes des interfaces Home ou Remote que chaque rôle de sécurité a le droit d'invoquer.

Les permissions de méthode sont définies par des relations entre l'ensemble des rôles de sécurité et l'ensemble des méthodes des interfaces Home et Remote, incluant également toutes les supers interfaces (incluant aussi les interfaces `javax.ejb.EJBHome` et `javax.ejb.EJBObject`). Une permission de méthode inclut la relation *(R, M)* si le rôle de sécurité *R* est autorisé à invoquer la méthode *M*.

L'assembleur de l'application définit les permissions de méthode dans le descripteur de déploiement en utilisant l'élément `method−permission` :

- Chaque élément `method−permission` comprend une liste de un ou plusieurs rôles de sécurité et une liste de une ou plusieurs méthodes. Tous les rôles de sécurité listés sont autorisés à invoquer toutes les méthodes de la liste. Chaque rôle de sécurité de la liste est identifié par l'élément `role−name` et chaque méthode est identifiée par l'élément `method`.
- La permission totale d'une méthode d'un EJB est constituée par l'union de toutes les permissions définies dans tous les éléments `method−permission` du descripteur de déploiement.
- Un rôle de sécurité ou une méthode peut apparaitre dans plusieurs éléments `method−permission`.

Il est possible que certaines méthodes ne soient assignées à aucun rôle de sécurité. Cela signifie que n'importe qui peut l'invoquer.

L'exemple suivant montre comment les rôles de sécurité sont assignés à des permissions de méthode dans le descripteur de déploiement :

```
...
<method-permission>
   <role-name>tomcat</role-name>
   <method>
      <ejb-name>Op</ejb-name>
      <method-name>*</method-name>
   </method>
</method-permission>
...
```

Gestion programmée de la sécurité

Comme les règles de sécurité ne peuvent pas toujours être déclarées de manière déclarative, l'architecture EJB fournit également une interface simple que le programmeur peut utiliser pour accéder au contexte de sécurité à partir de son

code.

L'interface `javax.ejb.EJBContext` fournit deux méthodes qui permettent au programmeur d'accéder aux informations de sécurité du client de l'EJB :

```
public interface javax.ejb.EJBContext {
   ...
   //
   // The following two methods allow the EJB class
   // to access security information
   //
   java.security.Principal getCallerPrincipal() ;
   boolean isCallerInRole (String roleName) ;
   ...
}
```

Utilisation de getCallerPrincipal()

L'objectif de la méthode `getCallerPrincipal()`est de permettre à une méthode d'un EJB d'obtenir le nom (principal name) du client de l'EJB. La méthode peut ensuite par exemple utiliser ce nom pour accéder à des informations de sécurité dans une base de donnée.

L'EJB peut obtenir ce nom en utilisant la méthode `getName()`de l'interface `java.security.Principal` retournée par `getCallerPrincipal()`.

Utilisation de isCallerInRole(String roleName)

L'objectif de la méthode `isCallerInRole(String roleName)` est de permettre au programmeur de coder les vérifications de sécurité qui ne peuvent pas être écrites de manière déclarative dans le descripteur de déploiement.

Le code de l'EJB utilise la méthode `isCallerInRole(String roleName)` pour tester si le client de la méthode endosse bien un rôle de sécurité particulier. Les rôles de sécurité sont définis par l'assembleur de l'application dans le descripteur de déploiement et sont assignés à des "Principals" (utilisateurs réels) par le déployeur.

Déclaration des rôles de sécurité référencés dans le code de l'EJB

Le programmeur du Bean doit déclarer un élément `security-role-ref` dans le descripteur de déploiement pour tous les noms de rôles de sécurité utilisés dans son code. Déclarer les références aux rôles de sécurités permet à l'assembleur de l'application ou au déployeur de lier ces noms de rôles de sécurité aux rôles de sécurités réels définis au niveau de l'application grâce aux éléments `security-role`.

```
   ...
   <enterprise-beans>
      ...
      <session>
         <ejb-nameOp</ejb-name>
```

```
      <ejb-class>sb.OpBean</ejb-class>
      ...
      <security-role-ref>
         <role-name>role1</role-name>
      </security-role-ref>
      ...
   </session>
   ...
</enterprise-beans>
...
```

Le descripteur de déploiement dans cet exemple indique que l'EJB `Op` utilise un code du type `isCallerInRole("role1")` dans au moins une de ses méthodes métier.

Lier les références de rôles de sécurités aux rôles de sécurités réels

Si les élément `security-role` ont été définis dans le descripteur de déploiement, toutes les références déclarées dans les éléments `security-role-ref` doivent être liées aux rôles définis dans les éléments `security-role`.

Le descripteur de déploiement suivant montre comment lier la référence de rôle de sécurité appelé `role1` au rôle de sécurité `tomcat`.

```
...
<enterprise-beans>
   ...
   <session>
      <ejb-name>Op</ejb-name>
      <ejb-class>sb.OpBean</ejb-class>
      ...
      <security-role-ref>
         <role-name>role1</role-name>
         <role-link>tomcat</role-link>
      </security-role-ref>
      ...
   </session>
   ...
</enterprise-beans>
...
```

En résumé, les noms des rôles utilisés dans le code EJB (lors de l'appel à la méthode isCallerInRole) sont en fait des références à des rôles de sécurité réels. Ceci rend le code de l'EJB indépendant de la configuration de sécurité décrite dans le descripteur de déploiement. La programmeur rend ces références disponibles à l'assembleur de l'application ou au déployeur grâce aux éléments `security-role-ref` inclus dans les éléments `session` ou `entity` du descripteur de déploiement.

Guide du programmeur EJB : Définir le descripteur de déploiement

Public visé et contenu

Ce guide est destiné au fournisseur de composants EJB, c'est−à−dire à la personne chargée de développer des composants logiciels coté serveur. Il explique comment écrire le descripteur de déploiement des EJBs.

Contenu du guide :

Principes

Le programmeur du Bean doit fournir un descripteur de déploiement associé aux EJB qu'il a écrit. L'assembleur de l'application peut ensuite l'adapter afin de fournir un descripteur de déploiement XML conforme à la DTD définie dans la spécification EJB 2.0 (voir `$JONAS_ROOT/xml/ejb−jar_2_0.dtd`).

Pour déployer des EJBs sur le serveur d'application, certaines informations manquantes au descripteur de déploiement standard peuvent être nécessaires. Par exemple les informations de mapping d'un Bean Entity CMP sur une base de donnée. Ces informations sont définies pendant le phase de déploiement dans un autre descripteur qui est spécifique à JOnAS. La DTD du descripteur de déploiement spécifique JOnAS est disponible dans `$JONAS_ROOT/xml/jonas−ejb−jar_X_Y.dtd`. Le nom du fichier du descripteur de déploiement spécifique à JOnAS doit être le nom du descripteur standard prefixé par 'jonas−'.

JOnAS interprète le tag <!DOCTYPE> lorsqu'il parse les descripteurs de déploiement. Le parser essaie d'abord de retrouver le fichier de DTD dans le CLASSPATH, ensuite il utilise l'url ou le chemin spécifié.

Dans l'exemple suivant, le parser retrouve le fichier de DTD dans le jar de JOnAS

```
<!DOCTYPE jonas−ejb−jar PUBLIC "−//ObjectWeb//DTD JOnAS 2.4//EN" "http://www.objectweb.org/jonas/
<!DOCTYPE jonas−ejb−jar SYSTEM "/usr/local/jonas/xml/jonas−ejb−jar_2_4.dtd">
```

Le descripteur de déploiement standard doit contenir certaines informations pour chaque EJB dont :

- le nom de l'EJB,
- la classe d'implémentation de l'EJB,
- l'interface d'accueil (home) de l'EJB,

- l'interface distante de l'EJB,
- le type de l'EJB,
- le type réentrant ou non de l'EJB,
- le type de gestion de l'état pour les Beans Sessions (avec ou sans état),
- le type de gestion des transactions pour les Beans Sessions,
- le type de gestion de la persistance pour les Beans Entité,
- la classe de clé primaire pour les Beans Entité,
- la liste des champs dont la persistance est gérée par le conteneur pour les Beans Entité CMP,
- les entrées dans l'environnement de l'EJB,
- les références aux autres EJBs,
- les références aux fabriques de connexion du serveur (message, JDBC, ...),
- les attributs transactionnels.

Le descripteur de déploiement spécifique contient des informations pour chaque EJB, dont :

- le nom JNDI sous lequel sera attaché l'interface d'accueil de l'EJB,
- le nom JNDI de l'objet DataSource utilisé par l'EJB,
- le nom JNDI de chaque référence aux autres EJBs,
- le nom JNDI des objets JMS utilisés par l'EJB,
- les informations sur le mapping de l'EJB avec la base de donnée, dans le cas d'un Bean Entité dont la persistance est gérée par le conteneur.

Exemple de descripteur pour un Bean Session

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN" "http://java.su
<ejb-jar>
  <description>Here is the description of the test's beans</description>
  <enterprise-beans>
    <session>
      <description>... Bean example one ...</description>
      <display-name>Bean example one</display-name>
      <ejb-name>ExampleOne</ejb-name>
      <home>tests.Ex1Home</home>
      <remote>tests.Ex1</remote>
      <ejb-class>tests.Ex1Bean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
      <env-entry>
        <env-entry-name>name1</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>value1</env-entry-value>
      </env-entry>
      <ejb-ref>
        <ejb-ref-name>ejb/ses1</ejb-ref-name>
        <ejb-ref-type>session</ejb-ref-type>
        <home>tests.SS1Home</home>
        <remote>tests.SS1</remote>
      </ejb-ref>
      <resource-ref>
```

```
        <res-ref-name>jdbc/mydb</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Application</res-auth>
      </resource-ref>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>ExampleOne</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
    <container-transaction>
      <method>
        <ejb-name>ExampleOne</ejb-name>
        <method-inter>Home</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Supports</trans-attribute>
    </container-transaction>
    <container-transaction>
      <method>
        <ejb-name>ExampleOne</ejb-name>
        <method-name>methodOne</method-name>
      </method>
      <trans-attribute>NotSupported</trans-attribute>
    </container-transaction>
    <container-transaction>
      <method>
        <ejb-name>ExampleOne</ejb-name>
        <method-name>methodTwo</method-name>
        <method-params><method-param>int</method-param></method-params>
      </method>
      <trans-attribute>Mandatory</trans-attribute>
    </container-transaction>
    <container-transaction>
      <method>
        <ejb-name>ExampleOne</ejb-name>
        <method-name>methodTwo</method-name>
        <method-params><method-param>java.lang.String</method-param></method-params>
      </method>
      <trans-attribute>NotSupported</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>


<!DOCTYPE jonas-ejb-jar PUBLIC "-//ObjectWeb//DTD JOnAS 2.4//EN" "http://www.objectweb.org/jonas/dtds/
<jonas-ejb-jar>
  <jonas-session>
    <ejb-name>ExampleOne</ejb-name>
    <jndi-name>ExampleOneHome</jndi-name>
```

```
    <jonas-ejb-ref>
      <ejb-ref-name>ejb/ses1</ejb-ref-name>
      <jndi-name>SS1Home_one</jndi-name>
    </jonas-ejb-ref>
    <jonas-resource>
      <res-ref-name>jdbc/mydb</res-ref-name>
      <jndi-name>jdbc_1</jndi-name>
    </jonas-resource>
  </jonas-session>
</jonas-ejb-jar>
```

Exemple de descripteur pour un Bean Entité avec persistance CMP (CMP 1.1)

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN" "http://java.su
<ejb-jar>
  <description>Here is the description of the test's beans</description>
  <enterprise-beans>
    <entity>
      <description>... Bean example one ...</description>
      <display-name>Bean example two</display-name>
      <ejb-name>ExampleTwo</ejb-name>
      <home>tests.Ex2Home</home>
      <remote>tests.Ex2</remote>
      <ejb-class>tests.Ex2Bean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>tests.Ex2PK</prim-key-class>
      <reentrant>False</reentrant>
      <cmp-version>1.x</cmp-version>
      <cmp-field>
        <field-name>field1</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>field2</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>field3</field-name>
      </cmp-field>
      <primkey-field>field3</primkey-field>
      <env-entry>
        <env-entry-name>name1</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>value1</env-entry-value>
      </env-entry>
    </entity>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>ExampleTwo</ejb-name>
        <method-name>*</method-name>
```

```
      </method>
      <trans-attribute>Supports</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

```
<!DOCTYPE jonas-ejb-jar PUBLIC "-//ObjectWeb//DTD JOnAS 2.4//EN" "http://www.objectweb.org/jonas/dtds/
<jonas-ejb-jar>
  <jonas-entity>
    <ejb-name>ExampleTwo</ejb-name>
    <jndi-name>ExampleTwoHome</jndi-name>
    <jdbc-mapping>
      <jndi-name>jdbc_1</jndi-name>
      <jdbc-table-name>YourTable</jdbc-table-name>
      <cmp-field-jdbc-mapping>
        <field-name>field1</field-name>
        <jdbc-field-name>dbf1</jdbc-field-name>
      </cmp-field-jdbc-mapping>
      <cmp-field-jdbc-mapping>
        <field-name>field2</field-name>
        <jdbc-field-name>dbf2</jdbc-field-name>
      </cmp-field-jdbc-mapping>
      <cmp-field-jdbc-mapping>
        <field-name>field3</field-name>
        <jdbc-field-name>dbf3</jdbc-field-name>
      </cmp-field-jdbc-mapping>
      <finder-method-jdbc-mapping>
        <jonas-method>
          <method-name>findByField1</method-name>
        </jonas-method>
        <jdbc-where-clause>where dbf1 = ?</jdbc-where-clause>
      </finder-method-jdbc-mapping>
    </jdbc-mapping>
  </jonas-entity>
</jonas-ejb-jar>
```

Astuces

Bien que certains caractères tels que le ">", puissent être utilisés, c'est une bonne pratique de les remplacer par des références à des entités XML.

La liste suivante présente les entités prédéfinies pour XML :

&lt;	<	less than
&gt;	>	greater than
&amp;	&	ampersand
&apos;	'	apostrophe

&quot;	"	quotation mark

EJB Programmer's Guide: EJB Packaging

Target Audience and Content

The target audience for this guide is the Enterprise Bean provider, i.e. the person in charge of developing the software components on the server side. It describes how the bean components should be packaged.

The content of this guide is the following:

Principles

Enterprise Beans are packaged for deployment in a standard Java programming language Archive file, called an *ejb−jar* file. This file must contain the following:

The beans' class files
: The class files of the remote and home interfaces, of the beans' implementations, of the beans' primary key classes (if there are any), and of all necessary classes.

The beans' deployment descriptor
: The ejb−jar file must contain the deployment descriptors, which are made up of:
  - ◊ The standard xml deployment descriptor, in the format defined in the EJB 2.0 specification. Refer to *$JONAS_ROOT/xml/ejb−jar_2_0.dtd*. This deployment descriptor must be stored with the name *META−INF/ejb−jar.xml* in the ejb−jar file.
  - ◊ The JOnAS−specific XML deployment descriptor in the format defined in *$JONAS_ROOT/xml/jonas−ejb−jar_X_Y.dtd*. This JOnAS deployment descriptor must be stored with the name *META−INF/jonas−ejb−jar.xml* in the ejb−jar file.

Example

Before building the ejb−jar file of the Account entity bean example, the java source files must be compiled to obtain the class files and the two XML deployment descriptors must be written.

Then, the ejb−jar file (OpEB.jar) can be built using the *jar* command:

```
cd your_bean_class_directory
mkdir META−INF
cp .../eb/*.xml META−INF
jar cvf OpEB.jar sb/*.class META−INF/*.xml
```

Guide du programmeur d'applications web

Public visé et Sommaire

Le public visé par ce guide sont les personnes en charge du développement des composants Web présents sur les parties serveurs. Ce guide décrit comment les développeurs de composants web doivent réaliser les descripteurs de déploiement associés à leurs composants web ainsi que la façon de les packager.

Le sommaire de ce guide est le suivant :

Développement de composants Web

Introduction

Un composant Web est un terme générique qui englobe à la fois les pages JSP et les Servlets. Ces composants Web sont packagés dans des fichiers `.war` et peuvent être déployés sur le serveur JOnAS via le service *conteneur web*. Ils peuvent également être intégrés à une application J2EE en englobant le fichier `.war` dans un fichier `.ear` (se référer au Guide du programmeur d'applications J2EE).

La distribution JOnAS inclut un exemple d'application Web : L'exemple EarSample.

Le structure arborescente de cette application est la suivante :

etc/xml	contient le fichier web.xml décrivant l'application web
etc/resources/web	contient les pages html et les images ; les pages JSP pouvant également être déposées ici.
src/org/objectweb/earsample/servlets	le code source des servlets
src/org/objectweb/earsample/beans	le code source des beans

Le répertoire contenant les beans n'est pas obligatoire si les beans utilisés proviennent d'une autre application.

Les pages JSP

Les Java Server Pages (JSP) sont une technologie permettant de mélanger des pages HTML statiques avec des pages HTML dynamiques écrites en Java afin d'encapsuler des traitements générant une partie du contenu. Se référer au document Java Server Pages™ et au Quickstart guide pour plus de détails.

Exemple:

L'exemple suivant illustre le fonctionnement d'une JSP listant le contenu d'un caddie.

```
<!-- Get the session -->
<%@ page session="true" %>

<!-- The import to use -->
<%@ page import="java.util.Enumeration" %>
<%@ page import="java.util.Vector"      %>

<html>
<body bgcolor="white">
  <h1>Content of your cart</h1><br>
  <table>
    <!-- The header of the table -->
    <tr bgcolor="black">
      <td><font color="lightgreen">Product Reference</font></td>
      <td><font color="lightgreen">Product Name</font></td>
      <td><font color="lightgreen">Product Price</font></td>
    </tr>

    <!-- Each iteration of the loop display a line of the table -->
    <%
      Cart cart = (Cart) session.getAttribute("cart");
      Vector products = cart.getProducts();
      Enumeration enum = products.elements();
      // loop through the enumeration
      while (enum.hasMoreElements()) {
          Product prod = (Product) enum.nextElement();
    %>
    <tr>
      <td><%=prod.getReference()%></td>
      <td><%=prod.getName()%></td>
      <td><%=prod.getPrice()%></td>
    </tr>
    <%
    } // end loop
    %>
  </table>
</body>
</html>
```

Une bonne pratique consiste à masquer tous les accès aux EJBs à partir de pages JSP en utilisant un bean mandataire,

référencé dans la page JSP grâce au tag `usebean`. Cette technique est illustrée dans l'exemple alarm, au sein duquel les fichiers .jsp communiquent avec les EJB via le bean mandataire ViewProxy.java.

Les Servlets

Les Servlets sont des modules java fonctionnant sur un serveur d'applications et répondant aux requêtes client. Les servlets ne sont pas liées à un protocole client−serveur spécifique. Cependant, elles sont la plupart du temps utilisées avec le protocole HTTP, et le terme "Servlet" est souvent utilisé en référence au terme "HTTP Servlet."

Les servlets utilisent les classes d'extension du standard Java présentes dans les packages `javax.servlet` (framework de base des Servlets) et `javax.servlet.http` (extensions du framework des servlets pour les servlets de réponse aux requêtes HTTP).

L'utilisation typique des servlets HTTP inclut :

- traiter et/ou stocker de données soumises via un formulaire HTML,
- fournir du contenu dynamique généré après exécution d'une requête sur la base de données,
- gérer les informations de la requête HTTP.

Pour plus de détails, se référer au document La technologie Servlet de Java™ et au tutoriel sur les servlets.

Exemple:

L'exemple suivant est un extrait d'une servlet listant le contenu d'un caddie.
Cet exemple est la version servlet de la page JSP vue précédemment.

```
import java.util.Enumeration;
import java.util.Vector;
import java.io.PrintWriter;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class GetCartServlet extends HttpServlet {

    protected void doGet(HttpServletRequest req, HttpServletResponse res)
                        throws ServletException, IOException {

        res.setContentType("text/html");
        PrintWriter out = res.getWriter();

        out.println("<html><head><title>Your cart</title></head>");
        out.println("<body>");
        out.println("<h1>Content of your cart</h1><br>");
        out.println("<table>");
```

```
        // The header of the table
        out.println("<tr>");
        out.println("<td><font color="lightgreen">Product Reference</font></td>");
        out.println("<td><font color="lightgreen">Product Name</font></td>");
        out.println("<td><font color="lightgreen">Product Price</font></td>");
        out.println("</tr>");

        // Each iteration of the loop display a line of the table
        HttpSession session = req.getSession(true);
        Cart cart = (Cart) session.getAttribute("cart");
        Vector products = cart.getProducts();
        Enumeration enum = products.elements();
        while (enum.hasMoreElements()) {
            Product prod = (Product) enum.nextElement();
            int prodId = prod.getReference();
            String prodName = prod.getName();
            float prodPrice = prod.getPrice();
            out.println("<tr>");
            out.println("<td>" + prodId + </td>);
            out.println("<td>" + prodName + </td>);
            out.println("<td>" + prodPrice + </td>);
            out.println("</tr>");
        }

        out.println("</table>");
        out.println("</body>");
        out.println("</html>");
        out.close();
    }
}
```

Accès à un EJB à partir d'une Servlet ou d'une page JSP

En utilisant le service conteneur web de JOnAS, il est possible d'accéder à un enterprise java bean (EJB) et à son environnement de manière conforme à la norme J2EE.

Le paragraphe suivant décrit :

1. Comment accéder à l'interface distante d'un bean.
2. Comment accéder à l'interface locale d'un bean.
3. Comment accéder à l'environnement d'un bean.
4. Comment démarrer les transactions au sein des servlets.

Note : Les exemples suivants sont extraits de l'exemple EarSample fourni avec la distribution JOnAS.

Accéder à l'interface distante d'un bean:

Dans cet exemple, la servlet récupère l'interface distante *OpHome* enregistrée dans JNDI en utilisant une référence EJB, puis elle crée une nouvelle instance du bean session :

```
import javax.naming.Context;
import javax.naming.InitialContext;

//remote interface
import org.objectweb.earsample.beans.secusb.Op;
import org.objectweb.earsample.beans.secusb.OpHome;

        Context initialContext = null;
        try {
            initialContext = new InitialContext();
        } catch (Exception e) {
            out.print("<li>Cannot get initial context for JNDI: ");
            out.println(e + "</li>");
            return;
        }
      // Connecting to OpHome thru JNDI
        OpHome opHome = null;
        try {
            opHome = (OpHome) PortableRemoteObject.narrow(initialContext.lookup("java:comp/env/ejb/Op
        } catch (Exception e) {
            out.println("<li>Cannot lookup java:comp/env/ejb/Op: " + e + "</li>");
            return;
        }
        // OpBean creation
        Op op = null;
        try {
            op = opHome.create("User1");
        } catch (Exception e) {
            out.println("<li>Cannot create OpBean: " + e + "</li>");
            return;
        }
```

Noter que les éléments suivants doivent être présents dans le fichiers `web.xml` se rapportant à l'application web:

```
  <ejb-ref>
    <ejb-ref-name>ejb/Op</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>org.objectweb.earsample.beans.secusb.OpHome</home>
    <remote>org.objectweb.earsample.beans.secusb.Op</remote>
    <ejb-link>secusb.jar#Op</ejb-link>
  </ejb-ref>
```

Accéder à l'interface locale d'un bean:

L'exemple suivant illustre comment obtenir une interface locale *OpLocalHome* en utilisant une référence locale à l'EJB :

```
//local interfaces
import org.objectweb.earsample.beans.secusb.OpLocal;
import org.objectweb.earsample.beans.secusb.OpLocalHome;
```

```
      // Connecting to OpLocalHome thru JNDI
        OpLocalHome opLocalHome = null;
        try {
            opLocalHome = (OpLocalHome)
                initialContext.lookup("java:comp/env/ejb/OpLocal");
        } catch (Exception e) {
            out.println("<li>Cannot lookup java:comp/env/ejb/OpLocal: " + e + "</li>");
            return;
        }
```

Ce qui suit est la partie correspondante dans le fichier `web.xml`:

```
  <ejb-local-ref>
    <ejb-ref-name>ejb/OpLocal</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local-home>org.objectweb.earsample.beans.secusb.OpLocalHome</local-home>
    <local>org.objectweb.earsample.beans.secusb.OpLocal</local>
    <ejb-link>secusb.jar#Op</ejb-link>
  </ejb-local-ref>
```

Accéder à l'environnement du composant:

Dans l'exemple qui suit, la servlet cherche à accéder à l'environnement du composant :

```
       String envEntry = null;
        try {
            envEntry = (String) initialContext.lookup("java:comp/env/envEntryString");
        } catch (Exception e) {
            out.println("<li>Cannot get env-entry on JNDI " + e + "</li>");
            return;
        }
```

Ce qui suit est la partie correspondante dans le fichier `web.xml` :

```
  <env-entry>
    <env-entry-name>envEntryString</env-entry-name>
    <env-entry-value>This is a string from the env-entry</env-entry-value>
    <env-entry-type>java.lang.String</env-entry-type>
  </env-entry>
```

Démarrer les transactions dans les servlets:

La servlet démarre les transactions via *UserTransaction*:

```
import javax.transaction.UserTransaction;

       // We want to start transactions from client: get UserTransaction
        UserTransaction utx = null;
        try {
            utx = (UserTransaction) initialContext.lookup("java:comp/UserTransaction");
        } catch (Exception e) {
```

```
        out.println("<li>Cannot lookup java:comp/UserTransaction: " + e + "</li>");
        return;
    }

    try {
        utx.begin();
        opLocal.buy(10);
        opLocal.buy(20);
        utx.commit();

    } catch (Exception e) {
        out.println("<li>exception during 1st Tx: " + e + "</li>");
        return;
    }
```

Définir le descripteur de déploiement web

Principes

Le développeur de composant web est responsable de la fourniture du descripteur de déploiement associé aux composants qu'il a développés. Lui et l'intégrateur de l'application sont responsables de fournir un descripteur de déploiement XML conforme à la DTD XML du descripteur de déploiement et respectant les spécifications version 2.3 des Servlets Java. (Se référer au fichier `$JONAS_ROOT/xml/web-app_2_3.dtd`).

Pour personnaliser les composants web, certaines informations non fournies dans le descripteur de déploiement XML standard peuvent être nécessaires. Il s'agit par exemple d'information portant sur le mapping entre le nom de la ressource référencée et son nom JNDI. Cette information peut être spécifiée durant la phase de déploiement, grâce à un autre descripteur de déploiement XML spécifique à JOnAS. Ce descripteur de déploiement spécifique aura une DTD XML située dans le fichier `$JONAS_ROOT/xml/jonas-web-app_X_Y.dtd`. Le nom du fichier de ce descripteur spécifique devra être construit en rajoutant au nom du descripteur de déploiement XML standard le préfixe '`jonas-`'.

JOnAS interprète le tag <!DOCTYPE> lors de l'analyse du contenu des fichiers descripteur de déploiement XML. L'analyse (parser) essaie tout d'abord d'obtenir la DTD spécifiée via le classpath, puis utilise l'URL spécifiée (ou path).

Dans les deux exemples suivants, l'analyse obtient en résultat le fichier `jonas-web-app_3_1.dtd` DTD via l'URL ou dans le répertoire /usr/local/jonas/xml/.

```
<!DOCTYPE jonas-web-app PUBLIC "-//ObjectWeb//DTD JOnAS Web App 3.1//EN" "http://www.objectweb.org
<!DOCTYPE jonas-web-app SYSTEM "/usr/local/jonas/xml/jonas-web-app_3_1.dtd">
```

Le descripteur de déploiement standard (web.xml) doit contenir les informations structurelles suivantes :

- La description de la servlet (incluant le nom, la classe ou le fichier JSP et les paramètres d'initialisation de la servlet),
- Les variables d'environnement en entrée,

- les références aux EJB,
- les références locales aux EJB,
- les références aux ressources,
- les références aux ressources d'environnement.

Le descripteur de déploiement spécifique à JOnAS (jonas−web.xml) doit contenir un certain nombre d'informations parmi lesquelles :

- Le nom JNDI des ressources externes référencées par le composant Web,
- Le nom JNDI des ressources d'environnement externes référencées par le composant Web,
- Le nom JNDI du bean référencé par le composant Web,
- Le nom de l'hôte virtuel sur lequel les servlets seront déployées,
- Le nom du contexte racine sur laquelle les servlets seront déployées,
- Un indicateur de compatibilité du classloader de l'application vis à vis du modèle de délégation java 2.

Elément <host> : Si le fichier de configuration du conteneur web contient des hôtes virtuels, l'hôte sur lequel le fichier WAR est déployé peut être spécifié.

Elément <context−root> : Le nom du context−root sur lequel l'application sera déployée devra être spécifié. Dans le cas contraire, le context−root peut être l'un des suivants :

- Si le war est packagé dans un fichier EAR, le context−root utilisé est celui spécifié dans le fichier application.xml.
- Si le war est en standalone, le contexte−root est le nom du fichier war (par exemple, le context−root est /jadmin pour jadmin.war).

Si le context−root est "/" ou vide, l'application web est déployée à la racine du contexte (par exemple http://localhost:8080/).

Elément <java2−delegation−model> : Indique la compatibilité avec le modèle de délégation java 2.

- Si `true` : le contexte de l'application web choisit un classloader suivant le modèle de délégation Java 2 (appel du classloader parent en premier).
- Si `false` : le classloader recherche d'abord dans l'application puis fait la demande au classloader parent.

Exemples de descripteurs de déploiement web

- Exemple de descripteur de déploiement web standard (web.xml):

```
<?xml version="1.0" encoding="ISO−8859−1"?>
<!DOCTYPE web−app PUBLIC "−//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
                         "http://java.sun.com/dtd/web−app_2_3.dtd">

<web−app>
  <servlet>
    <servlet−name>Op</servlet−name>
```

```
    <servlet-class>org.objectweb.earsample.servlets.ServletOp</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>Op</servlet-name>
  <url-pattern>/secured/Op</url-pattern>
</servlet-mapping>

<security-constraint>
  <web-resource-collection>
    <web-resource-name>Protected Area</web-resource-name>
      <!-- Define the context-relative URL(s) to be protected -->
      <url-pattern>/secured/*</url-pattern>
      <!-- If you list http methods, only those methods are protected -->
      <http-method>DELETE</http-method>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
      <http-method>PUT</http-method>
  </web-resource-collection>
  <auth-constraint>
    <!-- Anyone with one of the listed roles may access this area -->
    <role-name>tomcat</role-name>
    <role-name>role1</role-name>
  </auth-constraint>
</security-constraint>

<!-- Default login configuration uses BASIC authentication -->
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>Example Basic Authentication Area</realm-name>
</login-config>

<env-entry>
  <env-entry-name>envEntryString</env-entry-name>
  <env-entry-value>This is a string from the env-entry</env-entry-value>
  <env-entry-type>java.lang.String</env-entry-type>
</env-entry>

<!-- reference on a remote bean without ejb-link-->
<ejb-ref>
  <ejb-ref-name>ejb/Op</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>org.objectweb.earsample.beans.secusb.OpHome</home>
  <remote>org.objectweb.earsample.beans.secusb.Op</remote>
</ejb-ref>

<!-- reference on a remote bean using ejb-link-->
<ejb-ref>
  <ejb-ref-name>ejb/EjbLinkOp</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>org.objectweb.earsample.beans.secusb.OpHome</home>
  <remote>org.objectweb.earsample.beans.secusb.Op</remote>
  <ejb-link>secusb.jar#Op</ejb-link>
</ejb-ref>
```

```
  <!-- reference on a local bean -->
  <ejb-local-ref>
    <ejb-ref-name>ejb/OpLocal</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local-home>org.objectweb.earsample.beans.secusb.OpLocalHome</local-home>
    <local>org.objectweb.earsample.beans.secusb.OpLocal</local>
    <ejb-link>secusb.jar#Op</ejb-link>
  </ejb-local-ref>
</web-app>
```

- Exemple de descripteur de déploiement web spécifique (jonas-web.xml):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE jonas-web-app PUBLIC "-//ObjectWeb//DTD JOnAS Web App 3.1//EN"
                               "http://www.objectweb.org/jonas/dtds/jonas-web-app_3_1.dtd">
<jonas-web-app>
  <!-- Mapping between the referenced bean and its JNDI name, override the ejb-link
       if there is one in the associated ejb-ref in the standard Web Deployment Descriptor -->
  <jonas-ejb-ref>
    <ejb-ref-name>ejb/Op</ejb-ref-name>
    <jndi-name>OpHome</jndi-name>
  </jonas-ejb-ref>

  <!-- the virtual host on which deploy the web application -->
  <host>localhost</host>

  <!-- the context root on which deploy the web application -->
  <context-root>/web-application</context-root>
</jonas-web-app>
```

Trucs et astuces

Bien que les caractères tels que ">" soient valides, il est fortement recommandé de les remplacer par de références à des termes clés XML. La liste suivante donne les différents termes clés XML :

&lt;	<	plus petit que
&gt;	>	plus grand que
&amp;	&	ampersand
&apos;	'	apostrophe
&quot;	"	double quote

Les packages WAR

Les composants web sont packagés pour le déploiement dans un fichier d'archive standard Java appelé *war* (Web ARchive), qui est un *jar* similaire au package utilisé pour les bibliothèques de classes Java. Un fichier *war* a une structure arborescente spécifique. Le répertoire de plus haut niveau est le document racine de l'application.

Le document racine est celui contenant les pages JSP, les classes et archives de la partie client ainsi que les ressources web statiques. Le document racine contient le sous−répertoire appelé *WEB−INF* qui contient lui−même les fichiers et répertoires suivants :

- *web.xml*: Le descripteur de déploiement standard XML au format défini par les spécifications des servlets Java. Se référer au fichier *$JONAS_ROOT/xml/web−app_2_3.dtd*.
- *jonas−web.xml*: Le descripteur de déploiement optionnel spécifique à JOnAS au format défini dans le fichier *$JONAS_ROOT/xml/jonas−web_X_Y.dtd*.
- *classes*: un répertoire contenant les servlets et les classes utilitaires.
- *lib*: un répertoire contenant les archives jar des bibliothèques (bibliothèques de tags et toutes les bibliothèques utilitaires appelées par les classes situées sur la partie serveur). Si l'application web utilise des Entreprises Java Beans, il devra également contenir les *ejb−jars* afin de donner aux composants Web la visibilité sur les classes EJB.
  Cependant, si le fichier *war* a pour objectif d'être packagé dans un *ear*, les *ejb−jars* ne doivent pas être placés ici mais directement inclus dans l'*ear*. A cause de l'ordre de chargement des classes, les composants Web ont la visibilité sur les classes EJB.
  Des détails sur l'ordre de chargement des classes sont donnés dans le chapitre hiérarchie des chargeurs de classes JOnAS.

Exemple

Avant de fabriquer un fichier *war*, les fichiers source java doit être compilés pour obtenir les fichiers .class, situés dans le répertoire *WEB−INF/classes*, et les deux descripteurs de déploiement XML doivent être écrits.

Ensuite, le fichier *war* (<web−application>.war) est fabriqué en utilisant la commande *jar* :

```
cd <your_webapp_directory>
jar cvf <web−application>.war *
```

Durant le processus de développement, un version "non packagée" du fichier war peut être utilisée. Se référer à Comment utiliser les répertoires pour les applications web.

Guide du Programmeur de Connecteur J2EE

Le contenu de ce guide est le suivant :

Public auquel ce guide est destiné et son contenu

Ce chapitre est prévu pour les utilisateurs avancés de JOnAS concernés par l'EAI (Enterprise Application Integration) et utilisant les principes d'Architecture Connecteur J2EE, voir l' annexe pour une introduction aux connecteurs. Le public cible pour ce guide est le déployeur de connecteur et le programmeur. Il décrit le fichier de déploiement spécifique JOnAS (*jonas−ra.xml*) ainsi qu'un exemple de code utilisé pour accéder aux RARs déployés.

Principes

Les Connecteurs sont packagés pour le déploiement dans un fichier d'archive standard Java appelé un fichier *rar* (Resource ARchive), qui est décrit dans la spécification d'Architecture Connecteur J2EE.

La méthode standard de création du fichier *jonas−ra.xml* est d'utiliser l'outil RAConfig. Pour une description complète, voir RAConfig.

Definir le Descripteur de Déploiement de Connecteur JOnAS

Le fichier *jonas−ra.xml* contient des informations particulières à JOnAS décrivant des informations de déploiement, de logging, de pooling, de connexions jdbc, et des valeurs de propriétés de config RAR.

- *Balises de déploiement*:
  - jndiname: (Obligatoire) Nom sous lequel le fichier RAR sera enregistré. Cette valeur sera utilisée dans la section resource−ref d'un EJB.
  - rarlink: Jndiname d'un fichier RAR de base. Utile dans le cas d'un déploiement de plusieurs de connexions "factories" sans avoir à redéployer l'intégralité du fichier RAR. Quand ce mécanisme est utilisé, la seule entrée dans le fichier RAR est un fichier *META−INF/jonas−ra.xml*.
  - native−lib: Répertoire où devront être déployés des fichiers additionnels du RAR.
- *Balises de logging*:
  - log−enabled: Determine si le logging sera activé pour le RAR.
  - log−topic: Log topic à utiliser pour le log PrintWriter, qui permet un handler séparé pour chaque RAR déployé.

- *Balises de pooling*:
  - pool−init: Taille initiale du pool de connexions géré.
  - pool−min: Taille Minimum du pool de connexions géré.
  - pool−max: Taille Maximum du pool de connexions géré. La valeur −1 correspond à illimitée.
  - pool−max−age: Durée maximum (en millisecondes) de conservation d'une connexion dans le pool. La valeur 0 correspond à un temps illimité.
  - pstmt−max: Nombre maximum de requêtes pré−compilées (PreparedStatements) par connexion gérée dans le pool. Nécessaire uniquement avec le RA JDBC de JOnAS ou le rar d'un autre éditeur de base de données. La valeur 0 correspond à illimité et la valeur −1 invalide le cache.
- *Balises de Connexion*: Valide uniquement avec l'implémentation d'une connexion de type java.sql.Connection.
  - jdbc−check−level: Niveau de contrôle qui sera effectué pour la connexion jdbc. La valeur 0 correspond à aucun contrôle, 1 pour vérifier que la connexion n'est pas fermée avant de la retourner, et plus grand que 1 pour envoyer la requête jdbc−test−statement.
  - jdbc−test−statement: Requête SQL de test envoyée sur la connexion si la valeur jdbc−check−level est positionnée en conséquence.
- *Balises de valeurs de propriétés de config*:
  - Chaque entrée doit correspondre à la config−property specifiée dans le fichier *ra.xml* du fichier rar.

Exemples de Description de Déploiement

La partie ci−dessous d'un fichier *jonas−ra.xml* montre le lien à un fichier rar de base nommé BaseRar. Toutes les propriétés du fichier rar de base seront héritées et toutes les valeurs indiquées dans ce fichier *jonas−ra.xml* écraseront les autres valeurs.

```
<jonas−resource>
  <jndiname>rar1</jndiname>
  <rarlink>BaseRar</rarlink>
  <native−lib>nativelib</native−lib>
  <log−enabled>false</log−enabled>
  <log−topic>com.xxx.rar1</log−topic>
  <jonas−config−property>
    <jonas−config−property−name>ip</jonas−config−property−name>
    <jonas−config−property−value>www.xxx.com</jonas−config−property−value>
  </jonas−config−property>
  .
  .
</jonas−resource>
```

La partie suivante d'un fichier *jonas−ra.xml* file montre la configuration d'un fichier rar jdbc.

```
<jonas−resource>
  <jndiname>jdbc1</jndiname>
  <rarlink></rarlink>
  <native−lib>nativelib</native−lib>
  <log−enabled>false</log−enabled>
  <log−topic>com.xxx.jdbc1</log−topic>
  <pool−params>
```

```
      <pool-init>0</pool-init>
      <pool-min>0</pool-min>
      <pool-max>100</pool-max>
      <pool-max-age>0</pool-max-age>
      <pstmt-max>20</pstmt-max>
    </pool-params>
    <jdbc-conn-params>
      <jdbc_check-level>2</jdbc_check-level>
      <jdbc-test-statement>select 1</jdbc-test-statement>
    </jdbc-conn-params>
    <jonas-config-property>
      <jonas-config-property-name>url</jonas-config-property-name>
      <jonas-config-property-value>jdbc:oracle:thin:@test:1521:DB1</jonas-config-property-value>
    </jonas-config-property>
    .
    .
  </jonas-resource>
```

Packaging d'un Connecteur

Les Connecteurs sont packagés pour le déploiement dans un fichier d'archive standard java appelé fichier rar (Resource adapter ARchive). Ce fichier peut contenir les éléments suivants :

Descripteur de déploiement d'un Connecteur

Le fichier rar doit contenir les descripteurs de déploiement, qui sont constitués de:

- ◊ Le descripteur de déploiement standard xml, au format défini par la spécification J2EE 1.3. Voir *$JONAS_ROOT/xml/connector_1_0.dtd*. Ce descripteur de déploiement doit être enregistré sous le nom *META-INF/ra.xml* dans le fichier rar.
- ◊ Le descripteur de déploiement spécifique xml JOnAS au format défini dans la DTD *$JONAS_ROOT/xml/jonas-ra_X_Y.dtd*. Ce descripteur de déploiement de JOnAS doit être enregistré sous le nom *META-INF/jonas-ra.xml* dans le fichier rar.

Composants de Connecteur (jar)

Au moins un jar qui contient les interfaces java, l' implémentation, et les classes utilitaires exigées par le connecteur.

Bibliothèques natives spécifiques à la plate-forme

Bibliothèques natives utilisées par le connecteur.

Divers

Fichiers html, fichiers d'images, ou fichiers locaux utilisés par le connecteur.

Avant de déployer un fichier rar, le fichier xml spécifique de JOnAS doit être configuré et ajouté. Voir la **section RAConfig** pour plus d' information.

Utilisation et Déploiement d'un Connecteur

Avoir accès à un Connecteur implique de suivre les étapes suivantes :

- Le fournisseur de bean doit spécifier les éléments nécessaires à la fabrication de connexion en déclarant, dans son descripteur de déploiement d'EJB, une *référence de "factory" de connexions du connecteur*. Par exemple :

```
<resource-ref>
  <res-ref-name>eis/MyEIS</res-ref-name>
  <res-type>javax.resource.cci.ConnectionFactory</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

Le lien avec le véritable nom JNDI de la *factory de connexion* (ici `adapt_1`) est fait dans le descripteur de déploiement spécifique de JOnAS en utilisant l'élément suivant :

```
<jonas-resource>
  <res-ref-name>eis/MyEIS</res-ref-name>
  <jndi-name>adapt_1</jndi-name>
</jonas-resource>
```

Cela signifie que le progammeur de bean aura accès à une instance de la *factory de connexion* en utilisant l'interface JNDI via le nom *java:comp/env/eis/MyEIS* :

```
// obtain the initial JNDI naming context
Context inictx = new InitialContext();

// perform JNDI lookup to obtain the connection factory
javax.resource.cci.ConnectionFactory cxf =
        (javax.resource.cci.ConnectionFactory)
             inictx .lookup("java:comp/env/eis/MyEIS");
```

Le programmeur de bean peut ainsi obtenir une connexion en appelant la méthode `getConnection` sur la *factory de connexion*.

```
javax.resource.cci.Connection cx = cxf.getConnection();
```

L'instance de connexion retournée représente une prise au niveau applicatif sur la connexion physique utilisée pour accéder à l'EIS sous-jacent.
Après en avoir fini avec la connexion, elle doit être fermée en utilisant la méthode `close` sur l'interface `Connection` :

```
cx.close();
```

- ♦ Le Connecteur doit être déployé avant d'être utilisé par l'application. Déployer le connecteur nécessite les actions suivantes :
  - ◊ Créer un fichier de *configuration de connecteur spécifique de JOnAS* qui sera inclus dans le connecteur.

Ce fichier XML jonas−ra est utilisé pour configurer le connecteur en environnement opérationnel et reflète les valeurs de toutes les propriétés déclarées dans le descripteur de déploiement standard pour le connecteur, plus les propriétés de configuration spécifiques de JOnAS. JOnAS fournit un outil de déploiement **RAConfig** qui est capable de construire ce fichier XML à partir d'un descripteur de déploiement RA à l'intérieur d'un fichier RAR. Exemple :

```
RAConfig −j adap_1 ra
```

Ces propriétés peuvent être spécifiques à chaque connecteur et à son EIS sous−jacent. Elles sont utilisées pour configurer le connecteur via sa classe `managedConnectionFactory`. Il est obligatoire que cette classe fournisse les méthodes d'accès et de mise à jour ("getter" et "setter") pour chacune de ses propriétés (comme cela est requis dans le cadre de la spécification d'Architecture Connecteur).

Après avoir configuré le fichier jonas−ra.xml créé ci−dessus, il peut être ajouté au connecteur en exécutant la commande suivante :

```
RAConfig −u jonas−ra.xml ra
```

Cela ajoutera le fichier xml au fichier ra.rar, qui est maintenant prêt pour le déploiement.

- Le service JOnAS *resource* doit être configuré et démarré au lancement de JOnAS :
  Dans le fichier `jonas.properties` :
  - ◊ Vérifier que le nom `resource` est inclus dans la propriété `jonas.services`.
  - ◊ Utiliser une des méthodes suivantes pour déployer un fichier RAR :
    - · Le nom des fichiers *connecteur* (le suffixe '.rar' est optionnel) doit être ajouté dans la liste des Connecteurs devant être utilisés, dans la propriété `jonas.service.resource.resources`. Si le suffixe '.rar' n'est pas utilisé dans la propriété, il le sera quand il essaiera d'allouer le connecteur spécifié.

      ```
      jonas.service.resource.resources  MyEIS.rar, MyEIS1
      ```

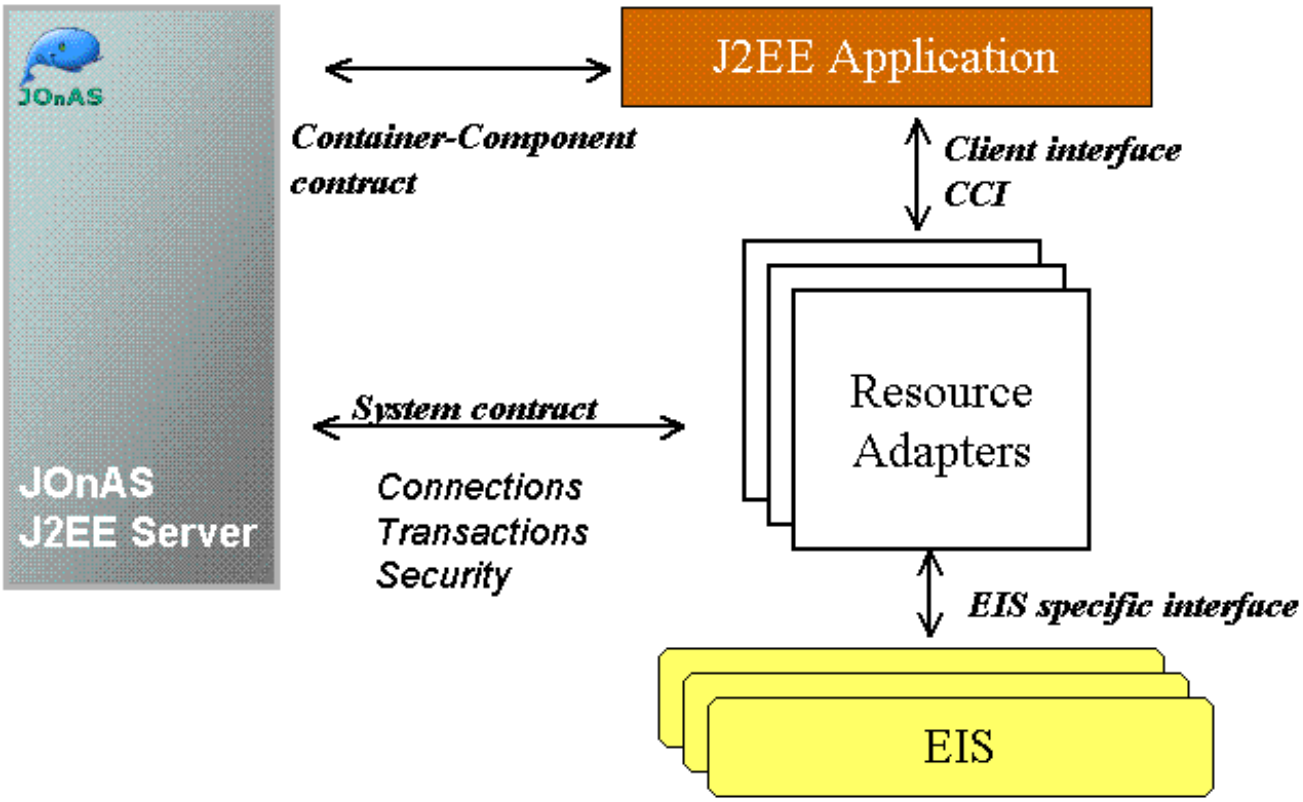
    - · Mettre le fichier RAR dans le répertoire autoload des connecteurs de $JONAS_BASE, la valeur par défaut est $JONAS_BASE/rars/autoload. A noter que le nom de ce répertoire peut varier si `jonas.service.resource.autoload` dans jonas.properties est configuré différemment.
    - · Déployer le RAR via la commande `jonas admin −a xxx.rar`.
    - · Déployer le RAR via la console d'administration JonasAdmin.

Connecteurs JDBC

Ces connecteurs génériques JDBC sont apportés par JOnAS et remplacent le service DBM. Voir **Configuration des Connecteurs JDBC** pour une description complète et un guide d'utilisation.

Annexe : Principes de l'Architecture J2EE Connecteur

L'Architecture Connecteur Java permet à un serveur d'application tel que JOnAS de se connecter à différents systèmes d'information de l'entreprise ("Enterprise Information Systems" ou EIS). elle définit pour les applications d'entreprise (basées sur des EJB, servlet, JSP ou clients J2EE) une manière de communiquer avec les EIS existants. Ceci nécessite l'utilisation d'un composant logiciel tiers appelé "Connecteur" pour chaque type d' EIS, qui devra être déployé au préalable sur le serveur d'application. Le Connecteur est un composant d'architecture comparable à un pilote logiciel, qui met en relation l' EIS, le serveur d'application, et l'application d'entreprise (composants J2EE dans le cas de JOnAS comme serveur d'application). Le connecteur fournit une interface (la "Common Client Interface" ou CCI) à l'application d'entreprise (composants J2EE) pour accéder à l'EIS. Il fournit également des interfaces standards pour se "plugger" au serveur d'application, de telle sorte qu'ils puissent collaborer et rendre les mécanismes de niveau système (transactions, sécurité, et gestion des connexions) transparents pour les composants applicatifs.



Le connecteur s'intègre avec JOnAS et fournit la connectivité entre l'EIS, JOnAS, et l' application. L'application fournit les traitements de la logique métier sur les données de l'EIS en utilisant l'API client du connecteur (CCI), alors que les transactions, les connexions (y compris le pooling), et la sécurité avec l' EIS sont gérées par JOnAS à travers le connecteur (system contract).

Guide du programmeur d'applications clientes J2EE

Public visé et sommaire

Le public visé par ce guide sont les personnes en charge du développement des composants applicatifs présents sur la partie cliente (tiers client). Il décrit comment les développeurs doivent fabriquer les descripteurs de déploiement qui s'appliqueront à leurs composants clients. Il décrit également comment doivent être packagés ces composants clients.

Le sommaire de ce guide est le suivant :

Démarrer une application J2EE client

Démarrer le client

L'application client J2EE est contenue dans un fichier .jar qui peut être :

- une application client standalone dans un fichier .jar,
- un client empaquetté dans un fichier .ear, un seul fichier ear pouvant contenir plusieurs clients java.

Tous les fichiers nécessaires au lancement du conteneur client sont contenus dans le fichier `JONAS_ROOT/lib/client.jar`. Ce jar inclut un fichier clairement identifié par le nom de la classe à lancer. Pour démarrer le conteneur client, taper simplement :

`java -jar $JONAS_ROOT/lib/client.jar -?`. Cette commande démarrera le conteneur client et affichera les informations concernant son utilisation.

Pour démarrer le conteneur client sur une machine distante, copier le fichier client.jar et invoquer le conteneur client en tapant l'instruction `java -jar path_to_your/client.jar.`

Le client qui doit être lancé par le conteneur client doit être donné en tant qu'argument du conteneur client.
Par exemple : `java -jar client.jar myApplication.ear`

ou `java -jar client.jar myClient.jar`.

Configurer le conteneur client

accès JNDI

Définir l'accès JNDI ainsi que le protocole à utiliser représente une partie importante de la configuration.
Le serveur JOnAS et le conteneur client utilisent les valeurs spécifiées dans le fichier de paramètres `carol.properties`.
Ce fichier peut être utilisé à différents niveaux.
Le fichier `carol.properties` est recherché en respectant les priorités suivantes, par ordre décroissant:

- `Le fichier carol.properties` spécifié par l'argument `-carolFile` du conteneur client
- le fichier `carol.properties` packagé dans l'application client (le jar client)
- si il n'a pas été trouvé selon les deux méthodes précédentes, le fichier `carol.properties` contenu dans le fichier `JONAS_ROOT/lib/client.jar` sera utilisé.

Un bonne façon de faire est de mettre à jour le fichier `carol.properties` de votre `client.jar` avec votre fichier `carol.properties` personnalisé.
par exemple : `jar -uf client.jar carol.properties`

Configuration de la fonction Trace

Le conteneur client `client.jar` inclut un fichier de paramètres nommé `traceclient.properties`. C'est le même fichier que celui présent dans le répertoire `JONAS_ROOT/conf`.
Un fichier de configuration différent peut être utilisé pour la fonction Trace en spécifiant le paramètre `-traceFile` lors de l'invocation du conteneur client.
Le fichier du `client.jar` peut également être remplacé avec la commande `jar -uf client.jar traceclient.properties`.

Spécifier le client à utiliser (cas de l'EAR)

Un ear peut contenir plusieurs clients java qui sont décrits dans les éléments `<module><java>` du fichier `application.xml`.
Pour invoquer le conteneur client avec un ear, tel que `java -jar client.jar my.ear`, vous aurez à spécifier le client java à utiliser si plusieurs clients existent. Sinon, le premier client sera utilisé par défaut.
Pour spécifier le client jar à utiliser à partir d'un ear, utiliser l'argument `-jarClient` en précisant le nom du client à utiliser.
L'exemple `earsample` fournis avec JOnAS possède deux clients java dans son ear.

Spécifier le répertoire pour dépackager l'ear (cas de l'EAR)

Par défaut, le conteneur client utilisera le paramètre système `java.io.tmpdir.`
Pour utiliser un autre répertoire temporaire, spécifier le chemin grâce à l'argument `-tmpDir`du conteneur client.

Exemples

Les exemples `earsample` et `jaasclient` fournis avec JOnAS sont packagés pour être utilisés par le conteneur client.
Le premier exemple illustre le cas d'un client à l'intérieur d'un ear. Le second exemple illustre le cas de l'utilisation d'un client standalone.

Définir le descripteur de déploiement client

Principes

Le développeur de composants clients est responsable de la fourniture du descripteur de déploiement associé aux composants qu'il aura développés.

Les responsabilités incombant aux développeurs de composants clients et aux intégrateurs d'applications sont de fournir un descripteur de déploiement XML conforme à la DTD XML du descripteur de déploiement telle que définie dans les spécifications Java ™ Version 1.3 pour les applications clientes. (Se référer au fichier `$JONAS_ROOT/xml/application-client_1_3.dtd`).

Pour personnaliser les composants clients, des informations non définies dans le descripteur de déploiement XML standard peuvent être nécessaires. De telles informations devront inclure, par exemple, le mapping entre le nom d'une ressource référencée et son nom JNDI. Cette information peut être spécifiée durant la phase de déploiement dans un autre descripteur de déploiement XML, spécifique à JOnAS. La DTD du descripteur de déploiement XML spécifique à JOnAS est situé dans le fichier `$JONAS_ROOT/xml/jonas-client_X_Y.dtd`. Le fichier du descripteur de déploiement XML spécifique à JOnAS doit impérativement s'appeler '`jonas-client.xml`'.

JOnAS interprète le tag <!DOCTYPE> lors de l'analyse (parsing) du fichier XML descripteur de déploiement.
La fonction de "parsing" essaie d'atteindre la DTD spécifiée via le classpath, puis utilise l'URL spécifiée (ou le chemin -path).

Dans les deux exemples suivants, la fonction de "parsing" atteint le fichier `jonas-client_3_2.dtd` DTD via l'URL ou dans le répertoire /usr/local/jonas/xml/.

```
<!DOCTYPE application-client PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application Client 1.3//
<!DOCTYPE application-client SYSTEM "/usr/local/jonas/xml/application-client_1_3.dtd">
```

Le descripteur de déploiement standard (application-client.xml) doit contenir les informations structurelles suivantes :

- Une description du Client,
- les variables d'environnement,
- références à des EJB,
- références aux ressources,
- références aux ressources d'environnement,

- le gestionnaire de procédure de rappel (CallbackHandler) à utiliser.

Le descripteur de déploiement spécifique à JOnAS (jonas−client.xml) doit contenir les informations suivantes :

- le nom JNDI des ressources externes référencées par le composant client,
- le nom JNDI des ressources externes d'environnement référencées par le composant client,
- le nom JNDI des beans référencés par le composant client,
- les aspects de sécurité incluant le fichier jaas, les variables en entrée du jaas et le couple login/password à utiliser pour chaque gestionnaire de procédure de rappel (CallbackHandler).

Exemples de descripteur de déploiement client

- Exemple d'un descripteur de déploiement client standard (application−client.xml):

```
<?xml version="1.0" encoding="UTF−8"?>
<!DOCTYPE application−client PUBLIC "−//Sun Microsystems, Inc.//DTD J2EE Application Client 1.3
                                    "http://java.sun.com/dtd/application−client_1_3.dtd">

<application−client>
  <display−name>Client of the earsample</display−name>
  <description>client of the earsample</description>

  <env−entry>
    <env−entry−name>envEntryString</env−entry−name>
    <env−entry−type>java.lang.String</env−entry−type>
    <env−entry−value>Test of envEntry of application−client.xml file</env−entry−value>
  </env−entry>


  <ejb−ref>
    <ejb−ref−name>ejb/Op</ejb−ref−name>
    <ejb−ref−type>Session</ejb−ref−type>
    <home>org.objectweb.earsample.beans.secusb.OpHome</home>
    <remote>org.objectweb.earsample.beans.secusb.Op</remote>
    <ejb−link>secusb.jar#EarOp</ejb−link>
  </ejb−ref>

  <resource−ref>
    <res−ref−name>url/jonas</res−ref−name>
    <res−type>java.net.URL</res−type>
    <res−auth>Container</res−auth>
  </resource−ref>

  <callback−handler>org.objectweb.jonas.security.auth.callback.LoginCallbackHandler</callback−h

</application−client>
```

- Exemple d'un descripteur de déploiement spécifique (jonas−client.xml) :

```
<?xml version="1.0" encoding="UTF−8"?>
<!DOCTYPE jonas−client PUBLIC "−//ObjectWeb//DTD JOnAS Client 3.2//EN"
                              "http://www.objectweb.org/jonas/dtds/jonas−client_3_2.dtd">
```

```
<jonas-client>

  <jonas-resource>
    <res-ref-name>url/jonas</res-ref-name>
    <jndi-name>http://jonas.objectweb.org</jndi-name>
  </jonas-resource>

  <jonas-security>
    <jaasfile>jaas.config</jaasfile>
    <jaasentry>earsample</jaasentry>
    <username>jonas</username>
    <password>jonas</password>
  </jonas-security>

</jonas-client>
```

Trucs et astuces

Bien que les caractères tel que ">" soient valides, il est fortement recommandé de les remplacer par des références à des termes clés XML.

La liste suivante donne les différents termes clés XML:

&lt;	<	plus petit que
&gt;	>	plus grand que
&amp;	&	ampersand
&apos;	'	apostrophe
&quot;	"	double quote

Package Client

Principes

Les composants clients sont packagés en vue du déploiement dans un fichier archive standard Java appelé fichier *jar* (Java ARchive). La racine de cette archive contient un sous−répertoire appelé *META−INF*, qui contient les fichiers et répertoires suivants :

- *application−client.xml*: Descripteur de déploiement XML standard au format défini dans les spécifications J2EE 1.3. Se référer au fichier *$JONAS_ROOT/xml/application−client_1_3.dtd*.

- *jonas−client.xml*: Descripteur de déploiement XML spécifique à JOnAS au format défini dans le fichier *$JONAS_ROOT/xml/jonas−client_X_Y.dtd*.

Le manifeste de ce jar client doit contenir le nom de la classe à lancer (contenant la méthode principale). Celle−ci est définie par la valeur de l'attribut *Main−Class* du fichier manifeste.
Dans le cas d'un client standalone (non empaquetté dans un Ear), toutes les classes des Ejb sur lesquels s'effectuent les consultations (excepté le skeleton) devront être incluses dans le *jar* .

Exemple

Deux exemples de fabrication de client java sont fournis.

- Le premier est donné par le build.xml de l'exemple `earsample` avec un client java à l'intérieur de l'ear.
  Se référer aux attributs client1jar et client2jar.
- Le second est donné par le build.xml de l'exemple `jaasclient` avec un client standalone java qui effectue des consultations sur un EJB.
  Se référer à l'attribut clientjars.

Guide de l'intégrateur d'application J2EE

Public visé et contenu

Le public visé pour ce guide regroupe les intégrateurs ("Application Assembler" dans la spécification) et les développeurs d'application, c'est−à−dire les personnes en charge de combiner un ou plusieurs composants (ejb−jars et/ou wars) afin de créer une application J2EE. Il décrit comment les composants J2EE doivent être packagés pour créer une application J2EE.

Le contenu de ce guide est le suivant :

Définir le descripteur de déploiement d'Ear

Principes

Le développeur d'applications est responsable de la fourniture du descripteur de déploiement associé à l'application qu'il a développée (Enterprise ARchive). L'intégrateur d'application a la responsabilité de fournir un descripteur de déploiement XML conforme à la DTD telle que définie dans les spécifications J2EE version 1.3. (se référer à `$JONAS_ROOT/xml/application_1_3.dtd`).

Pour déployer des applications J2EE sur le serveur d'applications, toutes les informations doivent être rassemblées dans un et un seul descripteur de déploiement contenu dans un fichier au format XML. Le nom du fichier XML descripteur de déploiement pour l'application est `application.xml` et doit être situé à la racine du répertoire META−INF.

JOnAS interprète le tag <!DOCTYPE> lors de l'analyse du fichier XML descripteur de déploiement.
JOnAS essaie tout d'abord d'obtenir la DTD spécifiée via le classpath, puis d'utiliser l'URL spécifiée (ou path).

Dans les deux exemples suivants, JOnAS obtient le fichier DTD `application_1_3.dtd` via l'URL ou dans le répertoire `/usr/local/jonas/xml/`.

```
<!DOCTYPE application PUBLIC '−//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN'
                             'http://java.sun.com/dtd/application_1_3.dtd'>
<!DOCTYPE application SYSTEM "/usr/local/jonas/xml/application_1_3.dtd">
```

Deux exemples d'applications J2EE sont fournis dans la distribution JOnAS :

- L'exemple EarSample
- L'exemple Alarm Demo

Le descripteur de déploiement standard doit contenir les informations structurelles suivantes :

- Composants EJB,
- Composants Web,
- Composants Client,
- Descripteur de déploiement alternatif pour ces composants,
- Rôle de sécurité.

Il n'existe pas de descripteur de déploiement spécifique à JOnAS pour les EAR (Enterprise ARchive).

Exemple simple de descripteur de déploiement d'application

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC '-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN'
                             'http://java.sun.com/dtd/application_1_3.dtd'>

<application>
  <display-name>Simple example of application</display-name>
  <description>Simple example</description>

  <module>
    <ejb>ejb1.jar</ejb>
  </module>
  <module>
    <ejb>ejb2.jar</ejb>
  </module>

  <module>
    <web>
      <web-uri>web.war</web-uri>
      <context-root>web</context-root>
    </web>
  </module>
</application>
```

Exemple avancé de descripteurs de déploiement (DD) d'application avec un DD alternatif et sécurité

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC '-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN'
                             'http://java.sun.com/dtd/application_1_3.dtd'>

<application>
```

```
  <display-name>Ear Security</display-name>
  <description>Application with alt-dd and security</description>
  <module>
    <web>
      <web-uri>admin.war</web-uri>
      <context-root>admin</context-root>
    </web>
  </module>
  <module>
    <ejb>ejb.jar</ejb>
    <alt-dd>altdd.xml</alt-dd>
  </module>
  <security-role>
    <role-name>admin</role-name>
  </security-role>
</application>
```

Trucs et astuces

Bien que les caractères tel que ">" soient valides, il est fortement recommandé de les remplacer par des références à des termes clés XML.

La liste suivante donne les différentes références à des termes clés XML :

&lt;	<	plus petit que
&gt;	>	plus grand que
&amp;	&	ampersand
&apos;	'	apostrophe
&quot;	"	double cote

Packager un EAR

Selon les standards Java, pour se déployer, les applications J2EE sont packagées dans un fichier d'archive appelé un fichier *ear* (Enterprise ARchive). Ce fichier peut contenir les informations suivantes :

Les composants web (war)
: Un ou plusieurs wars contiennent les composants web de l'application J2EE. En raison de l'ordre de chargement des classes, lorsque les wars d'une application J2EE sont packagés, il n'est pas nécessaire de packager les classes d'éventuels EJBs accédés par les composants Web dans le répertoire *WEB-INF/lib*. Des détails sur l'ordre de chargement de classes sont disponibles au chapitre Hiérarchie des chargeurs de classes JOnAS.

Les composants EJB (ejb-jar)
: Un ou plusieurs ejb-jars, qui contiennent les beans de l'application J2EE.

Les bibliothèques (jar)
: Un ou plusieurs jars qui contiennent les bibliothèques (que ce soit les bibliothèques de tags ou toutes bibliothèque d'utilitaires) utilisées par l'application J2EE.

Le descripteur de déploiement J2EE
: Le descripteur de déploiement standard XML au format défini dans les spécifications J2EE 1.3. Se référer au fichier *$JONAS_ROOT/xml/application_1_3.dtd*. Ce descripteur de déploiement doit être enregistré avec le nom *META−INF/application.xml* dans le fichier ear.

Exemple

Avant de générer un fichier ear pour une application J2EE, les ejb−jars et les wars qui doivent être packagés dans l'application J2EE devront être générés et le descripteur de déploiement XML (application.xml) aura été écrit.

Alors, le fichier ear (`<j2ee−application>.ear`) pourra être généré en utilisant la command *jar*:

```
cd <your_j2ee_application_directory>
jar cvf <j2ee−application>.ear *
```

Guide de déploiement et d'installation

Public visé

Le public visé par ce guide sont les personnes en charge du déploiement des applications.

Le contenu de ce guide est le suivant :

Principes des processus de déploiement et d'installation

Le déploiement et l'installation d'Enterprise Beans

Ce guide suppose que le développeur d'Enterprise Bean ait consulté le guide du programmeur d'Enterprise Beans et ait packagé les classes bean avec le descripteur de déploiement dans un fichier ejb−jar. Pour déployer des Entreprise Beans non packagés, se référer au Guide "Configurer le Service Conteneur EJB".

Pour déployer des Enterprise Beans dans JOnAS, la personne en charge du déploiement doit ajouter les classes d'interposition interfaçant les composants EJB avec les services fournis par le serveur d'applications JOnAS. L'outil GenIC fourni avec la distribution JOnAS permet de générer les classes d'interposition et de mettre à jour le fichier ejb−jar.
La personne en charge du déploiement de l'application peut également avoir besoin de modifier les descripteurs de déploiement afin de les adapter aux spécificités de l'environnement opérationnel. Cela devra être fait avant d'utiliser GenIC.

La personne en charge du déploiement peut choisir de déployer les Enterprise Beans en tant que composants d'une application stand alone, auquel cas le fichier ejb−jar doit être installé dans le répertoire `$JONAS_ROOT/ejbjars`. Elle pourra également les inclure dans un package war ou ear en suivant les instructions données ci−après.

Le déploiement et l'installation d'applications Web et J2EE

Une fois le package des composants d'applications complété, tel que décrit dans les guides Les Packages WAR or Packager un EAR, le fichier d'archive obtenu devra être installé dans le répertoire :

- `$JONAS_ROOT/webapps` pour les fichiers war
- `$JONAS_ROOT/apps` pour les fichiers ear

Exemple de déploiement et d'installation d'EJB en utilisant un fichier ejb−jar

Pour cet exemple, il est supposé que l'utilisateur veuille personnaliser le déploiement du bean *AccountImpl* de l'application `examples/src/eb` en changeant le nom de la table dans la base de données utilisée pour la persistance de *AccountImpl*.

Le répertoire actuel est `$JONAS_ROOT/examples/src/eb`. L'utilisateur doit procéder comme suit:

- **Editer** le fichier `jonas−ejb−jar.xml` et modifier la valeur de l'élément `<jdbc−table−name>` inclus dans l'élément `<jdbc−mapping>` correspondant à l'entité `AccountImpl`.
- **Compiler** tous les fichiers `.java` présents dans ce répertoire :
  ```
  javac −d ../../classes Account.java AccountImplBean.java
  AccountExplBean.java AccountHome.java ClientAccount.java
  ```
- **Effectuer le déploiement**
  - Générer un fichier ejbjar appelé `ejb−jar.jar` avec toutes les classes correspondantes et les deux descripteurs de déploiement :
    ```
    mkdir −p ../../classes/META−INF
    cp ejb−jar.xml ../../classes/META−INF/ejb−jar.xml
    cp jonas−ejb−jar.xml ../../classes/META−INF/jonas−ejb−jar.xml
    cd ../../classes
    jar cvf eb/ejb−jar.jar META−INF/ejb−jar.xml
    META−INF/jonas−ejb−jar.xml eb/Account.class eb/AccountExplBean.class
    eb/AccountHome.class eb/AccountImplBean.class
    ```
  - A partir du répertoire des sources, exécuter l'outil **GenIC** qui générera le fichier `ejb−jar.jar` final avec les classes d'interposition :
    ```
    GenIC −d ../../classes ejb−jar.jar
    ```
- **Installer** le fichier ejb−jar dans le répertoire `$JONAS_ROOT/ejbjars` :
  ```
  cp ../../classes/eb/ejb−jar.jar $JONAS_ROOT/ejbjars/ejb−jar.jar
  ```

Le serveur d'applications JOnAS peut désormais être démarré en utilisant la commande :
```
jonas start
```

Les étapes qui viennent d'être décrites pour la fabrication du nouveau fichier `ejb−jar.jar` expliquent le processus de déploiement. Il est généralement mis en oeuvre par un script de fabrication ANT.

Si *Apache ANT* est installé sur votre machine, tapez *ant install* dans le répertoire `$JONAS_ROOT/examples/src` afin de fabriquer et d'installer l'ensemble des fichiers `ejb−jar.jar` contenant les exemples.

Lors de l'écriture d'un fichier `build.xml` pour ANT, utiliser la tâche `ejbjar`, qui est une des tâches EJB optionnelles définies dans ANT 1.5. La tâche `ejbjar` contient un élément appelé `jonas`, qui implémente le processus de déploiement décrit précédemment (génération des classes d'interposition et mise à jour du fichier EJB−JAR).
Généralement la version la plus à jour de la tâche EJB contenant une implémentation récente de l'élément `jonas` est

fournie avec JOnAS, dans le répertoire `$JONAS_ROOT/lib/common/ow_jonas_ant.jar`. Cliquer ici pour obtenir la documentation correspondante de l'élément jonas.
L'extrait de code suivant provenant de l'exemple `$JONAS_ROOT/examples/src/alarm/build.xml` illustre ce qui vient d'être vu :

```
  <!-- ejbjar task -->
  <taskdef name="ejbjar"
     classname="org.objectweb.jonas.ant.EjbJar"
     classpath="${jonas.root}/lib/common/ow_jonas_ant.jar" />

  <!-- Deploying ejbjars via ejbjar task -->
  <target name="jonasejbjar"
    description="Build and deploy the ejb-jar file"
    depends="compile" >
   <ejbjar basejarname="alarm"
      srcdir="${classes.dir}"
      descriptordir="${src.dir}/beans/org/objectweb/alarm/beans"
      dependency="full">
    <include name="**/alarm.xml"/>
    <dtd publicId="-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
      location="${jonas.root}/xml/ejb-jar_2_0.dtd" />
    <dtd publicId="-//ObjectWeb//DTD JOnAS 2.5//EN"
      location="${jonas.root}/xml/jonas-ejb-jar_2_5.dtd" />
    <support dir="${classes.dir}">
      <include name="**/ViewProxy.class"/>
    </support>
    <jonas destdir="${dist.ejbjars.dir}" jonasroot="${jonas.root}"
orb="${objectweb_orb}" />
   </ejbjar>
  </target>
```

Déploiement et installation d'une application Web

Avant de déployer une application Web sur le serveur d'applications JOnAS, vous devez d'abord ***packager*** ses composants dans un fichier war en suivant les instructions fournies dans le chapître Les packages WAR.
Pour *Apache ANT*, se référer au paramètre war présent dans le fichier `$JONAS_ROOT/examples/earsample/build.xml`.

Ensuite, **installer** le fichier war dans le répertoire `$JONAS_ROOT/webapps`.
Note : Faites attention à ne pas installer le fichier war dans le répertoire `$CATALINA_HOME/webapps`.

Puis, vérifier la **configuration** : Avant d'exécuter l'application web, vérifier que le service *web* est présent dans le paramètre `jonas.services`. Le service *ejb* peut également être nécessaire si l'application Web fait appel à des enterprise beans.
Le nom du fichier war peut être ajouté dans la section `jonas.service.web.descriptors`.

Au final, **démarrer** le serveur d'applications :

```
jonas start
```

Les composants web sont déployés dans le container web créé lors du démarrage.

Si le fichier war n'a pas été ajouté à la liste `jonas.service.web.descriptors`, les composants web peuvent être dynamiquement déployés en utilisant la commande `jonas admin` ou l'utilitaire d'administration `JonasAdmin` tool.

Déploiement et installation d'une application J2EE

Avant de déployer une application J2EE sur le serveur d'applications JOnAS, vous devez d'abord **packager** ses composants dans un fichier ear en suivant les instructions fournies dans le guide Packager un EAR.
Pour *Apache ANT*, se référer au paramètre ear présent dans le fichier `$JONAS_ROOT/examples/earsample/build.xml`.

Ensuite, **installer** le ficher ear dans le répertoire `$JONAS_ROOT/apps`.

Puis, vérifier la **configuration**: avant d'exécuter l'application, vérifier que les services *ejb*, *web* et *ear* sont présents dans le paramètre `jonas.services`.
Le nom du fichier ear peut être ajouté dans la section `jonas.service.ear.descriptors`.

Au final, **démarrer** le serveur d'applications:

```
jonas start
```

Les composants de l'application sont déployés dans les conteneurs d'EJB et web créés lors du démarrage.

Si le fichier ear n'a pas été ajouté dans la liste `jonas.service.ear.descriptors`, les composants de l'application peuvent être dynamiquement déployés en utilisant la commande `jonas admin` ou l'utilitaire d'administration `JonasAdmin`.

Guide d'Administration

Le public auquel ce guide est destiné est l'administrateur du serveur JOnAS.

JOnAS fournit les deux outils suivants pour réaliser les tâches d'administration sur un serveur JOnAS en cours d'exécution :

- **`jonas admin`**, un outil en mode ligne de commande
- **`JonasAdmin`**, un outil graphique basé sur l'environnement Struts et la technologie JMX
    - Installation de JonasAdmin
    - Utilisation de JonasAdmin

Ces outils permettent également d'administrer un ensemble de serveurs JOnAS. Chaque serveur JOnAS est identifié par un nom, qui est la valeur de l'option `-n` utilisée dans la commande `jonas start` (le nom par défaut est `jonas`).
De plus, du fait que l'implémentaton open source de MX4J est intégrée dans JOnAS, l'administrateur du serveur est à même d'utiliser la console d'administration générique MC4J.

jonas admin

jonas admin est décrit au chapitre Commandes JOnAS.

JonasAdmin

Ce chapitre fournit des informations sur comment installer, configurer, et utiliser la console d'administration *JonasAdmin*.

JonasAdmin est le nouvel outil d'administration de JOnAS et remplace l'outil devenu obsolète *Jadmin*.

JonasAdmin a été développé en utilisant le framework Struts; il utilise des technologies standard telles que les Servlets Java et les Pages JavaServer (JSP). JonasAdmin offre une meilleure ergonomie que Jadmin et fournit des facilités d'administration intégrées pour le serveur Tomcat en fonctionnement incorporé dans JOnAS.

Installation de JonasAdmin

Conçu en temps qu'application web, JonasAdmin est packagé dans un fichier WAR et installé sous le répertoire `JONAS_ROOT/webapps/autoload/`. Ce fichier WAR peut être installé dans `JONAS_BASE/webapps/autoload` si une variable JONAS_BASE a été définie dans l'environnement. S'il est installé dans le répertoire `autoload`, JonasAdmin est déployé au lancement du serveur JOnAS, de ce fait la console d'administration est accessible de façon automatique.

De même que pour n'importe quelle application web, JonasAdmin a besoin d'un serveur de servlet pour être installé. De plus, le serveur JOnAS où tourne JonasAdmin doit avoir le service conteneur web présent dans la liste des services

définis dans le fichier de configuration `jonas.properties`.

Pour utiliser JonasAdmin, l'administrateur doit s'identifier et s'authentifier.
Le fichier de configuration `jonas-realm.xml` contient une définition de domaine de mémoire appelée `memrlm_1`, qui est référencée dans les deux fichiers de configuration `server.xml` et `jetty.xml`. Les deux valeurs par défaut du nom de l'utilisateur (***jonas*** ou ***jadmin***) et du mot de passe (***jonas***) correspondant au rôle `admin` peuvent être modifiées à ce niveau.

Utilisation de JonasAdmin

Une fois lancée, JonasAdmin peut administrer le serveur JOnAS dans lequel il tourne, aussi bien que les autres serveurs JOnAS avec lesquels il partage le même "registry". Typiquement, ceci est utilisé pour administrer des serveurs JOnAS fonctionnant sans le service de conteneur WEB.
A noter que les serveurs JOnAS administrés peuvent tourner sur un ou plusieurs serveurs physiques. De plus, si Tomcat est utilisé comme implémentation du service de conteneur WEB, il pourra être administré au moyen de JonasAdmin.

Pour un bon fonctionnement de JonasAdmin

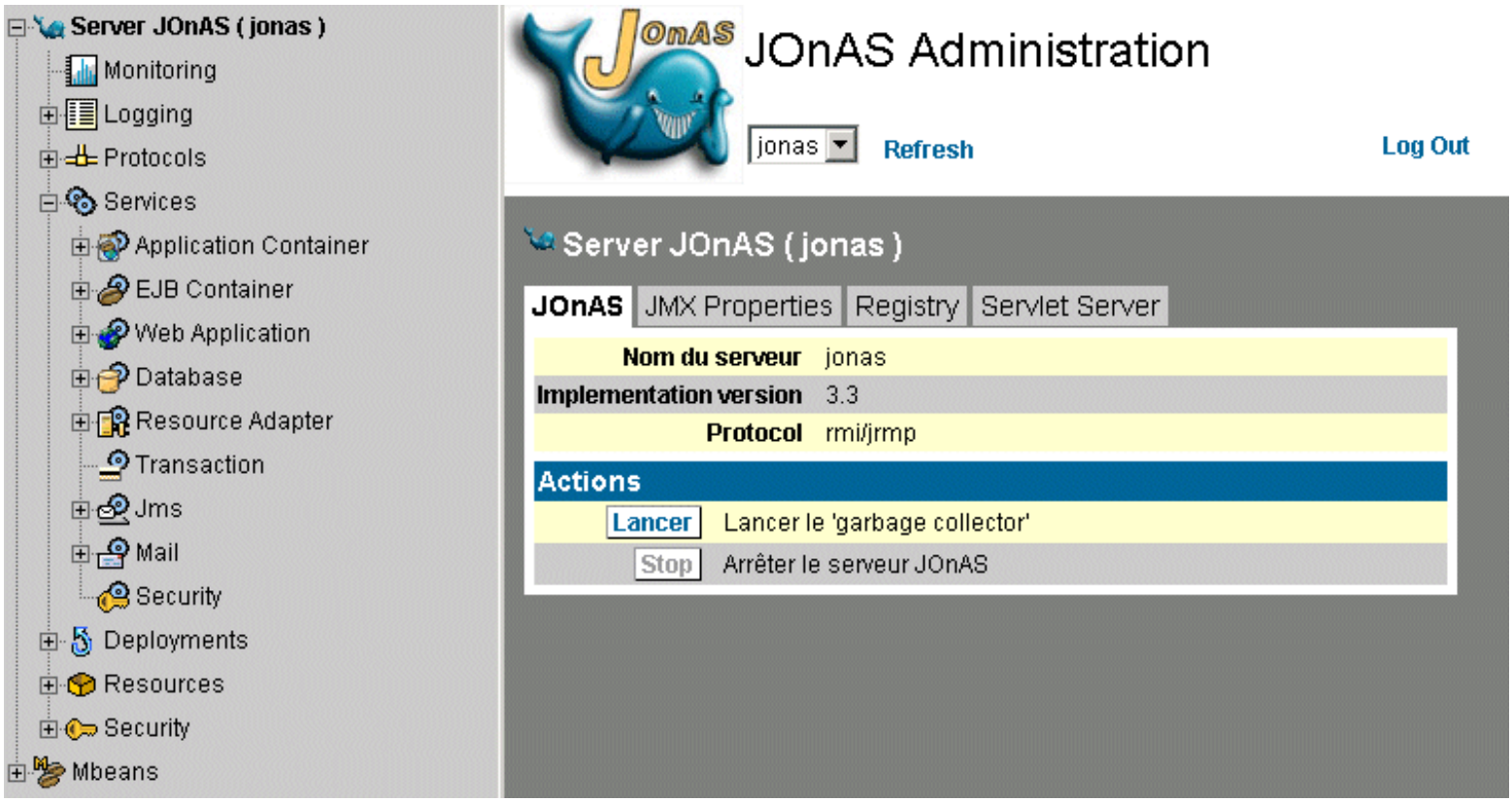
S'assurer que le service `web` est listé dans la propriété `jonas.services` dans le fichier de configuration `jonas.properties`. Dans le cas ou le package JOnAS utilisé ne comprend ni Tomcat ni Jetty (installés par ailleurs), en fonction du conteneur de Servlet utilisé, la variable d'environnement `CATALINA_HOME` ou `JETTY_HOME` doit avoir été positionnée au préalable. A noter que, si le conteneur de Servlet tourne sur Unix, la variable d'environnement `DISPLAY` doit être positionnée de façon à utiliser les possibilités de "monitoring" du serveur JOnAS de l'outil JonasAdmin.

Une fois que JOnAS est lancé, JonasAdmin doit être chargé s'il n'a pas été installé dans le répertoire `autoload`. La console d'administration est accessible à l'URL: `http://<hostname>:<portnumber>/jonasAdmin/` en utilisant n'importe quel navigateur web.

<*hostname*> est le nom de l'hôte où tourne le conteneur de Servlets et <*portnumber*> est le numéro de port http (8080 par défaut).

Une fois connecté, le cadre gauche de la page d'Accueil montre l'arborescence d'administration associée au serveur JOnAS où tourne l'application d'administration (l'exemple suivant indique que son nom est *jonas*). Si d'autres serveurs tournent en partageant le même "registry" que *jonas*, on peut utiliser la fenêtre en haut du cadre pour sélectionner le serveur à administrer.

Cliquer sur le noeud `Server JOnAS` pour afficher la page suivante:



L'arborescence d'administration dans cet écran permet d'accéder aux possibilités de gestion principales suivantes:

- Information générale concernant le serveur administré
- Surveillance du serveur
- Gestion des logs
- Gestion des protocoles de communication
- Présentation et configuration des services actifs
- Déploiement dynamique des modules applicatifs
- Gestion des ressources
- Gestion de la sécurité

La console permet également de visualiser les MBeans enregistrés dans le serveur de MBean et qui sont associés au serveur JOnAS administré concerné.

Gestion de Serveur

Affiche des information générales sur le serveur JOnAS administré, y compris le serveur JMX et le serveur WEB, et offre la possibilité de lister le contenu du "registry".

Surveillance de Serveur

Affiche l'utilisation de la mémoire, le nombre de threads créés par JOnAS, ainsi que d'autres informations de surveillance concernant les services et ressources gérées.

Gestion des Logs

Permet à l'administrateur de configurer le système de logs de JOnAS. De plus, si Tomcat est utilisé en tant qu'implémentation de service conteneur WEB, il permet la création des logs d'accès HTTP.

Gestion des protocoles de communication

Cette possibilité d'administration a trait à l'intégration de la gestion de Tomcat dans JonasAdmin.
Elle présente actuellement les connecteurs définis dans la configuration de Tomcat et permet la création de nouveaux connecteurs HTTP, HTTPS, ou AJP.
A noter que le sous−arbre `Protocols` n'est pas présenté si on utilise Jetty comme implémentation de service conteneur WEB.

Présentation des services actifs et configuration

Tous les services actifs ont un sous−arbre correspondant dans l'arbre `Services`.

La gestion des différents services de conteneurs consiste à présenter des informations sur les composants déployés dans ces conteneurs. De nouveaux composants peuvent être déployés en utilisant les possibilités de déploiement dynamique présentées dans la section suivante. Cependant, il peut s'avérer nécessaire de créer un nouveau contexte pour les composants WEB (package WAR) à déployer dans un serveur Tomcat avant l'étape de déploiement, si un contexte personnalisé est requis par le composant. Cette opération est réalisée en actionnant le bouton `New web application`.

D'une façon similaire, les services qui permettent la gestion des différents types de ressources (DataSources, Connecteurs, ressources Jms et Mail) fournissent également des informations sur les ressources déployées. De plus, les ressources déployées (DataSources ou MailFactories) peuvent être reconfigurées et leur nouvelle configuration rendue persistante en utilisant le bouton `Save`.

La gestion du service de transaction permet sa reconfiguration (qui peut être persistante) et présente des informations de surveillance sur les transactions gérées par JOnAS.

Déploiement dynamique avec JonasAdmin

Une fonction de gestion très utile est la possibilité de charger des composants J2EE autonomes (JAR, WAR, packages RAR) ou applications J2EE (packages EAR) en utilisant le sous−arbre `Deployments` dans la console JonasAdmin. La tâche de l'administrateur est facilitée par l'affichage de la liste des modules déployables, la liste des modules déployés, et la possibilité de transférer des modules d'une liste à une autre. Les modules déployables sont ceux installés dans les répertoires spécifiques à leur type. Par exemple, les fichiers JARs déployables sont les fichiers JARs non déployés installés dans `JONAS_BASE/ejbjars/` ou dans `JONAS_BASE/ejbjars/`*`autoload/`*.

Gestion des Ressources

Le sous−arbre `Resources` offre la possibilité de charger ou de créer de nouvelles ressources gérées par les services actifs. Par exemple, si le service JMS est actif, le sous−arbre JMS dans `Resources` présente les destinations JMS existantes (Topics et Queues), et permet le retrait des destinations inutilisées ainsi que la création de nouvelles destinations JMS.
L'ajout ou le retrait de ressources implique de reconfigurer le service correspondant. Si la nouvelle configuration est sauvée en utilisant le bouton `Save`, le fichier de configuration de JOnAS sera mis à jour. Ainsi dans l'exemple du service JMS, les destinations de type Topic retirées sont supprimées de la liste affectée à la propriété `jonas.service.jms.topics` et les destinations de type Topic nouvellement créées sont rajoutées à cette liste.

Gestion de la sécurité

Le sous−arbre `Security` présente les domaines de sécurité existants et permet la création de nouveaux domaines de différents types: domaines mémoire, base de données, et ldap.

Remarque concernant les possibilités de reconfiguration persistante

Il est important de remarquer que JOnAS et Tomcat ont des approches différentes en ce qui concerne la persistance de la reconfiguration. Dans JOnAS, toute opération `Save` est liée à une reconfiguration de service ou de ressource. Par exemple, l'administrateur peut reconfigurer un service et une ressource, mais il peut choisir de ne sauver que la nouvelle configuration de la ressource.
Dans Tomcat, l'opération `Save` est globale et concerne toutes les modifications de configuration qui ont été réalisées. Par exemple, si un nouveau connecteur HTTP est reconfiguré et un nouveau contexte créé pour une application web, les deux modifications de configuration seront sauvées si on utilise le bouton `Save`.

Guide de référence des commandes JOnAS

Les commandes fournies avec JOnAS sont décrites dans ce chapitre.

jonas
: Gestionnaire de JOnAS

jclient
: Lancement de client JOnAS

newbean
: Générateur de Bean

registry
: Lancement du "regisrty" d'objets distants Java

GenIC
: Générateur de classes de conteneur

JmsServer
: Lancement du serveur JMS

RAConfig
: Outil de configuration de connecteur

EJBServer
: Serveur JOnAS (commande obsolète)

JonasAdmin
: Serveur d'Administration de JOnAS (commande obsolète)

CheckEnv
: Vérificateur de configuration de JOnAS (commande obsolète)

jonas

Synopsis

`jonas start [-fg | -bg | -win] [-n name]`
: lance un serveur JOnAS

`jonas stop [-n name]`
: arrête un serveur JOnAS

`jonas admin [-n name] [admin_options]`
: administre un serveur JOnAS

`jonas check`
: vérifie l'environnement JOnAS

`jonas version`
: imprime la version de JOnAS

Description

Cette commande remplace les commandes obsolètes EJBServer, JonasAdmin, et CheckEnv. Elle offre

la possibilité de lancer, d'arrêter, ou d'administrer des serveurs JOnAS.

Le résultat de ce programme peut dépendre du répertoire depuis laquelle la commande est exécutée (existence d'un fichier *jonas.properties* dans le répertoire courant). Il est possible de modifier des propriétés système pour ce programme en utilisant la variable d'environnement **JAVA_OPTS**, si nécessaire. A noter que la mise à jour des propriétés système avec l'option −D sera prioritaire par rapport aux propriétés décrites dans les autres fichiers *jonas.properties* .

Les deux scripts suivants peuvent être consultés et modifiés pour résoudre des problèmes ou obtenir de l'information additionnelle :

jonas pour les systèmes UNIX
jonas.bat pour les systèmes WINDOWS

Il y a cinq sous−commandes différentes, qui dépendent du premier argument obligatoire:

jonas start
Lance un nouveau serveur JOnAS. Le processus peut être lancé au premier−plan, en arrière−plan ou dans une nouvelle fenêtre. Si on choisit l'option arrière−plan (option par défaut), le contrôle est rendu à celui qui a fait l'appel seulement quand le serveur est prêt. Le nom par défaut est **jonas**. Un nom différent peut être donné en utilisant l'option **−n**.

jonas stop
Arrête un serveur JOnAS en cours d'exécution. Utiliser l'option **−n** si le nom donné au serveur n'est pas celui par défaut.

jonas admin
Administre un serveur JOnAS. Utiliser l'option **−n** si le nom donné au serveur n'est pas celui par défaut. Utilisée sans aucune autre option, cette commande affiche à l'utilisateur un message d'invite à taper une commande d'administration (mode interactif). Chaque commande d'administration existe dans un mode non interactif, pour les utiliser dans des shell scripts ou des scripts de fichiers bat, par exemple. Se référer à la liste d'options pour une description de chacune d'entre elles. Une autre façon d'administrer JOnAS est de le faire avec l'outil graphique JonasAdmin. La fonctionnalité de ***JonasAdmin*** est essentiellement la même que celle de ***jonas admin***.

jonas check
Vérifie l'environnement avant de lancer un serveur JOnAS.

jonas version
Imprime la version courante de JOnAS.

Options

Chaque option peut n'être pertinente que pour un sous−ensemble des cinq sous−commandes. Par exemple, ***jonas check*** et ***jonas version*** n'acceptent aucune option.

−n name
Donne un nom au serveur JOnAS. Celui−ci est ***jonas*** par défaut . Utilisé avec ***start***, ***stop***, ou ***admin***.

−fg

Utilisé avec ***start*** uniquement. Le serveur est lancé en premier−plan : Le contrôle n'est rendu à l'utilisateur qu'à la fin du processus.

−bg

Utilisé avec ***start*** uniquement. Le serveur est lancé en arrière−plan. Le contrôle n'est rendu à l'utilisateur que quand le serveur JOnAS est prêt. C'est le mode par défaut.

−win

Utilisé avec ***start*** uniquement. Le serveur est lancé dans une nouvelle fenêtre.

−?

Utilisé avec ***admin*** uniquement. Imprime une aide avec toutes les options possibles.

−a filename

Utilisé avec ***admin*** uniquement. Déploit une nouvelle application décrite par *filename* à l'intérieur du serveur JOnAS. L'application peut être l'une des suivantes :

- un fichier standard *ejb−jar*. Cela va conduire à la création d'un nouveau conteneur EJB dans le serveur JOnAS. Si le nom de fichier a un chemin relatif, ce chemin est relatif à l'emplacement d'où a été lancé le serveur EJB ou au répertoire *$JONAS_ROOT/ejbjars* pour un fichier ejb−jar.
- un fichier standard *.war* contenant un composant WEB. Si le nom de fichier a un chemin relatif, ce chemin est relatif à l'emplacement où le serveur EJB a été lancé ou au répertoire *$JONAS_ROOT/webapps* pour un fichier war.
- un fichier standard *.ear* contenant une application J2EE complète. Si le nom de fichier a un chemin relatif, ce chemin est relatif à l'emplacement d'où le serveur EJB a été lancé ou au répertoire *$JONAS_ROOT/apps* pour un fichier ear.

−r filename

Utilisé avec ***admin*** uniquement. Supprime de façon dynamique une commande précédente ***−a filename***.

−gc

Utilisé avec ***admin*** uniquement. Lance le ramasse−miettes dans le serveur JOnAS considéré.

−passivate

Utilisé avec ***admin*** uniquement. Rend passives toutes les instances de beans entités. Ceci n'affecte que les instances hors transaction.

−e

Utilisé avec ***admin*** uniquement. Liste les propriétés du serveur JOnAS considéré.

−j

Utilisé avec ***admin*** uniquement. Liste les noms JNDI enregistrés, tels qu'ils sont vus par le serveur JOnAS considéré.

−l

Utilisé avec ***admin*** uniquement. Liste les beans chargés par le serveur JOnAS considéré.

−sync

Utilisé avec ***admin*** uniquement. Synchronise les instances des beans sur le serveur JOnAS considéré. A noter que ceci n'affecte que les instances qui ne sont pas impliquées dans une transaction.

−debug topic

Utilisé avec ***admin*** uniquement. Force le niveau de "topic" de trace à la valeur DEBUG.

−tt timeout

Utilisé avec ***admin*** uniquement. Modifie le timeout par défaut pour les transactions. *timeout* est exprimé en secondes.

Chaque option ***jonas admin*** a son équivalent dans le mode interactif. Pour entrer en mode interactif et accéder à la liste suivante de sous−commandes, taper ***jonas admin [−n name]*** sans aucun autre argument. Pour sortir du mode interactif, utilier la commande exit.

commande interactive	**commande correspondante en ligne**
addbeans	**−a fileName**
env	**−e**
gc	**−gc**
help	**−?**
jndinames	**−j**
listbeans	**−l**
removebeans	**−r fileName**
sync	**−sync**
trace	**−debug topic**
ttimeout	**−tt timeout**
quit	exit interactive mode

Exemples

```
jonas check
jonas start −n jonas1
jonas admin −n jonas1 −a bean1.jar
jonas stop −n jonas1
```

jclient

Synopsis

jclient [options] java−class [args]

Description

La commande *jclient* permet à l'utilisateur de lancer facilement un client java "lourd" qui pourra atteindre des beans sur des serveurs JOnAS distants et lancer des transactions distribuées.

Ceci n'est pas la façon conforme à J2EE de lancer un client java, qui consiste à packager le client java dans un conteneur client J2EE (voir Package Client).
La commande *jclient* pourrait être retirée dans une prochaine version.

Options

−cp classpath
Rajoute un classpath avant d'exécuter le programme java.

Exemples

```
jclient package.javaclassname args
```

newbean

Synopsis

newbean

Description

L'outil *newbean* aide le développeur de bean au début de son développement en générant les squelettes pour tous les fichiers nécessaires à la création d'un bean. A noter que cet outil ne crée que les modèles des fichiers. Ces fichiers doivent être personnalisés et il faut écrire la logique métier. Cependant, ces fichiers devraient être compilables.

Pour créer ces modèles, taper *newbean* et renseigner un ensemble de paramètres dans le mode interactif.

Le **Bean Name** doit commencer par une lettre majuscule. Eviter les noms réservés : Home, EJB, Session, Entity. Ce nom sera utilisé comme préfixe pour tous les noms de fichiers relatifs à ce bean.

Le **Bean Type** doit être l'un des suivants :

- **S** Session bean
- **E** Entity bean
- **MD** Message−Driven bean

Le **Session Type** doit être l'un des suivants:

- **L** Stateless Session Bean (Bean sans état de session)
- **F** Stateful Session Bean (Bean avec état de session)

Le **Persistence manager** doit être l'un des suivants :

- **C** Container−Managed Persistence

- **B** Bean−Managed Persistence

Le **Bean Location** doit être l'un des suivants :

- **R** Remote Interfaces
- **L** Local Interfaces

Le **Package Name** est un ensemble de chaînes de caractères séparées par des points représentant le package auquel le bean appartient. C'est généralement le même que le répertoire courant.

L'argument **Jar Name** est le nom qui sera utilisé pour construire le fichier .jar. Ne pas fournir l'extension ".jar" avec cet argument. Typiquement, c'est la dernière partie du nom du package qui est utilisée.

La **primkey** est la classe représentant la clé primaire, nécessaire uniquement pour les entity beans. Les valeurs possibles sont:

- **s** java.lang.String
- **i** java.lang.Integer
- **o** Object (sera choisie plus tard)

Exemple

```
newbean
Bean Name
> MyEntity

Bean type
  S   Session bean
  E   Entity bean
  MD  Message−Driven bean
> E

Persistance manager
  C  Container
  B  Bean
> C

Bean location
  R  Remote
  L  Local
> R

Package name
> truc.machin

Jar name
> machin

Primary Key class
0  S  String
  I  Integer
```

```
  O  Object
> S

Création du bean MyEntity (type ECR) dans le package truc.machin
Vos fichiers bean ont été créés. Vous pouvez maintenant les personnaliser.
```

registry

Synopsis

registry [<port>]

Description

L'outil *registry* crée et lance un objet distant "registry" sur le port spécifié de l'hôte courant, basé sur le type d' *ORB* défini dans la configuration JOnAS (*RMI* ou *Jeremie*).

Si le port est omis, le registry est lancé sur le port 1099 sur *RMI*, ou sur le port 1234 sur *Jeremie*.

A noter que, par défaut, le registry est lancé dans la même JVM que le serveur JOnAS. Dans ce cas, il n'est d'ailleurs pas nécessaire d'utiliser cet outil, le registry est lancé automatiquement.

Options

port

Numéro de Port.

Exemple

La commande *registry* peut être normalement exécutée en arrière−plan :

- `registry &` sur Unix, ou
- `start registry.bat` sur Windows

CheckEnv

Synopsis

CheckEnv [Options]

Cette commande est obsolète. Elle a été remplacée par jonas check.

Description

L'outil CheckEnv vérifie la cohérence de la configuration JOnAS et affiche le contenu des fichiers de configuration qui sont utilisés.

Options

−help

Affiche le mode d'utilisation de la commande CheckEnv.

Example

```
CheckEnv
```

Environment

Le résultat de cette commande peut dépendre du répertoire courant d'où elle est lancée.
(Existence d'un fichier *jonas.properties* dans le répertoire courant, ou existence du chemin . (dot) dans le classpath)

EJBServer

Synopsis

EJBServer [Options]

Cette commande est obsolète. elle a été remplacée par jonas start.

Description

Lance le serveur JOnAS.

Options

−D<property>=<value>

Toutes les propriétés JOnAS qui peuvent être définies dans le fichier *jonas.properties* peuvent également être positionnées par le biais de l'option standard *−D* de l'outil *java*.

Exemple

La commande *EJBServer* peut être normallement exécutée en arrière−plan :

- `EJBServer &` sur Unix, ou
- `start EJBServer` sur Windows.

Environnement

Le résultat de cette commande peut dépendre de la bibliothèque courante d'où elle est lancée. (Existence d'un fichier *jonas.properties* dans la bibliothèque courante.)

GenIC

Synopsis

GenIC [Options] <InputFileName>

Description

L'utilitaire GenIC génère les classes de conteneur pour JOnAS à partir des Enterprise Java Beans donnés.

L'argument *InputFileName* est soit le nom d'un fichier ejb−jar ou le nom de fichier d'un descripteur XML de déploiement de beans.

L'utilitaire GenIC réalise les opérations suivantes dans l'ordre indiqué :

1) génère les sources des classes de conteneur pour tous les beans définis dans le descripteur de déploiement,
2) compile ces classes via le compilateur java,
3) génère les stubs et les skeletons pour ces objets distants via le compilateur rmi, et
4) si le fichier *InputFile* est un fichier ejb−jar, ajoute les classes générées dans le fichier ejb−jar.

Options

−d *directory*

Spécifie le répertoire racine de la hiérarchie de classes.
Cette option peut être utilisée pour spécifier un répertoire cible pour les fichiers générés.
Si l'option *−d* n'est pas utilisée, la hiérarchie du package de la classe cible est ignorée et les fichiers générés sont placées dans le répertoire courant.

Si le fichier *InputFile* est un fichier ejb−jar, les classes générées sont ajoutées au fichier ejb−jar, à moins que l'option *−noaddinjar* ne soit spécifiée.

−javac *options*
Spécifie le nom du compilateur *java* à utiliser (*javac* par défaut).

−javacopts *options*
Spécifie les options à passer au compilateur *java*.

−keepgenerated
Ne supprime pas immédiatement les fichiers générés.

−noaddinjar
Si le fichier *InputFile* est un fichier ejb−jar, ne rajoute pas les classes générées au fichier ejb−jar.

−nocompil
Ne compile pas les fichiers sources générés via les compilateurs java et rmi.

−protocols
liste de protocoles séparés par des virgules (choisis parmi jeremie, jrmp, iiop, cmi) pour lesquels les stubs devraient être générés. Par défaut, `jrmp,jeremie`.

−rmiopts *options*
Spécifie les options à passer au compilateur *rmi* .

−verbose
Affiche des informations supplémentaires sur l'exécution de la commande.

−mappernames
Liste de noms des adapteurs (mappers), séparés par des virgules, pour lesquels les classes de conteneur vont être générées. Utilisé pour l'implémentation de la CMP 2.0 basée sur JORM. Un adapteur est utilisé par JORM pour accéder à une base de données spécifiée. Cette liste d'adapteurs correspond à la liste des bases de données potentielles sur lesquelles les "beans" entités peuvent être déployés.

Exemple

```
GenIC −d ../../classes sb.xml
```
génère les classes de conteneur de tous les Enterprise JavaBeans définis dans le fichier *sb.xml*. Les classes sont générées dans le répertoire *../../classes* conformément à la hiérarchie des classes.

```
GenIC sb.jar
```
génère les classes de conteneur pour tous les Enterprise JavaBeans définis dans le fichier *sb.jar* et rajoute les classes générées au fichier ejb−jar.

Environnement

Si *InputFile* est un descripteur de déploiement XML, le "classpath" doit inclure les chemins des répertoires dans lesquelles on peut trouver les classes "Enterprise Beans", ainsi que le chemin du répertoire spécifié dans l'option *−d*.

Si *InputFile* est un fichier ejb−jar, le "classpath" doit inclure le chemin du répertoire spécifié dans l'option *−d*.

JmsServer

Synopsis

JmsServer

Description

Lance le Serveur Joram (c'est−à−dire le MOM) avec les options par défaut.

Options

aucune

Example

La commande *JmsServer* est exécutée en règle générale en arrière−plan :

- `JmsServer &` sur Unix, ou
- `start JmsServer` sur Windows.

JonasAdmin

Cette commande est obsolète. Elle a été remplacée par jonas admin.

Description Générale

JonasAdmin est un outil qui réalise quelques fonctions d'administration sur un serveur JOnAS en fonctionnement.

Cet outil présente deux modes d'utilisation : un *mode commande et un mode interactif.*

La commande **`JonasAdmin`** sans argument (ou avec l'unique option **`−n <jonas−name>`**) exécute l'outil dans le mode interactif.

Les fonctions effectuées par l'outil *JonasAdmin* sont résumées dans le tableau ci−dessous :

Functionnalité	*Mode Commande*	*Mode Interactif*
Crée de façon dynamique un nouveau conteneur JOnAS et y ajoute les beans.	**`−a fileName`**	***a***ddbeans
Affiche la liste des propriétés du serveur JOnAS.	**`−e`**	***e***nv
Exécute le ramasse−miettes dans le serveur JOnAS.	**`−gc`**	***g***c
Affiche l'aide à l'utilisation de la commande JonasAdmin.	**`−?`** or **`−h`**	***h***elp

Affiche la liste des noms JNDI déclarés.	**-j**	***j***ndinames
Affiche la liste des types de beans chargés actuellement par le serveur JOnAS.	**-l**	***l***istbeans
Identifie un serveur EJB.	**-n jonasName**	***n***ame
Quitte JonasAdmin.	non applicable	***q***uit
Retire de façon dynamique un conteneur JOnAS ainsi que tous ses beans.	**-r fileName**	***r***emovebeans
Stoppe le serveur JOnAS.	**-s**	***s***top
Synchronise les instances "entity bean" sur le serveur JOnAS.	**-sync**	***sy***nc
Liste ou met à jour le niveau de trace dans le serveur JOnAS.	**-t topic**	***tr***ace
Met à jour le timeout par défaut des transactions (en secondes) dans le serveur JOnAS.	**-tt timeout**	***tt***imeout

Par défaut (sans l'option **-n <jonas-name>**), *JonasAdmin* accède au serveur dont le nom est **jonas**.

L'utilisation détaillée de cet outil est fournie dans la suite de ce document.

Synopsis

1. JonasAdmin
2. JonasAdmin −n jonas−name
3. JonasAdmin [−n jonas−name] [options]

Description

La première option (1.), exécute *JonasAdmin* dans le mode interactif, sur le serveur JOnAS dont le nom est `jonas.`

La deuxième (2.), éxécute *JonasAdmin* dans le mode interactif, sur le serveur JOnAS dont le nom est `jonas-name`.

La troisième (3.), exécute *JonasAdmin* en mode commande, la tâche réalisée dépend de l'option sélectionnée.

Options

addbeans <beans-file-name>
: Crée de façon dynamique un nouveau conteneur JOnAS.
Le *beans−file−name* est un fichier ejb−jar.
Si le nom de fichier possède un chemin relatif, ce chemin est soit relatif à l'emplacement d'où on a lancé le serveur JOnAS, soit relatif au répertoire *$JONAS_ROOT/ejbjars* pour un fichier ejb−jar.

env
: Liste les propriétés du serveur JOnAS.

gc
Exécute le ramasse−miettes dans le serveur JOnAS considéré.
help
Donne un résumé des options.
jndinames
Liste les noms JNDI déclarés.
listbeans
Liste les beans actuellement chargés par le serveur JOnAS.
name
Change le serveur JOnAS par défaut sur lequel vont s'appliquer les commandes suivantes.
quit
Quitte JonasAdmin.
removebeans <beans−file−name>
Retire de façon dynamique le conteneur JOnAS identifié par *beans−file−name*.
stop
Stoppe le serveur JOnAS.
sync
Synchronise les instances "entity bean" sur le serveur JOnAS courant. A noter que cela n'affecte que les instances qui ne sont pas impliquées dans une transaction.
trace
Liste les filtres de trace dans le serveur JOnAS courant.
trace <topic>
Met le niveau de trace (topic) à DEBUG.
ttimeout <seconds>
Modifie le timeout par défaut pour les transactions.

Exemples

JonasAdmin −j

Liste les noms JNDI enregistrés actuellement par le serveur JOnAS.

RAConfig

Synopsis

RAConfig [Options] <InputFileName> [<OutputFileName>]

Description

L'utilitaire RAConfig génère un *fichier de configuration de connecteur spécifique à JOnAS* `(jonas−ra.xml)` à partir d'un fichier `ra.xml` (Descripteur de déploiement de Connecteur).

Le *InputFileName* est le nom de fichier d'un connecteur.

Le *OutputFileName* est le nom de fichier d'un connecteur "output" utilisé avec l'option −p (obligatoire) ou −u (optionnel).

Options

−? ou **−help**

Donne un résumé des options.

−dm,−ds,−pc,−xa *DriverManager, DataSource, PooledConnection, XAConnection*
Spécifie la valeur "rarlink" à configurer, utilisée avec l'option −p.

−j *jndiname*
C'est une option obligatoire. Elle précise le nom JNDI de la "*connection factory"*. Ce nom correspond au nom de l'élément <jndi−name> de l'élément <jonas−resource> dans le descripteur de déploiement spécifique de JOnAS. Ce nom est utilisé par le service *resource* pour enregistrer dans JNDI la *connection factory* correspondant au connecteur.

−p *database properties file*
Spécifie le nom du fichier database.properties à traiter. Le résultat de ce traitement sera un fichier jonas−ra.xml qui va mettre à jour le fichier `/META−INF/jonas−ra.xml` dans le rar en sortie.

−r *rarlink*
Spécifie le nom jndi d'un fichier rar à mettre en correspondance. Cette option pourra être utilisée quand ce fichier rar va hériter de tous les attributs associés au nom jndi spécifié. Si cette option est spécifiée dans le fichier `jonas−ra.xml`, c'est le seul fichier nécessaire dans le rar, et le fichier `ra.xml` sera traité à partir du fichier rarlink.

−u *inputname*
Spécifie le nom du fichier XML à traiter. Ce fichier va mettre à jour le fichier `/META−INF/jonas−ra.xml` dans le rar. Si cet argument est utilisé, c'est le seul argument exécuté.

−verbose
mode Verbeux. Affiche le descripteur de déploiement du connecteur sur une sortie `System.out standard`.

Exemple

- **`RAConfig −j adapt_1 MyRA.rar`**
  Génère le fichier `jonas−ra.xml` à partir du fichier *ra.xml*.

  Après que le jonas−ra.xml ait été configuré pour le fichier rar MyRA
  `RAConfig −u jonas−ra.xml MyRA.rar`
  Met à jour /insére le fichier `jonas−ra.xml` dans le fichier rar.
- **`RAConfig −dm −p MySQL1 $JONAS_ROOT/rars/autoload/JOnAS_jdbcDM MySQL_dm`**

  Génère le fichier `jonas−ra.xml` à partir du fichier *ra.xml* de JOnAS_jdbcDM.rar et insère les valeurs correspondantes à partir du fichier MySQL1.properties. Le fichier

`jonas-ra.xml` est ainsi ajouté/mis à jour au fichier MySQL_dm.rar. Ce fichier rar peut ainsi être déployé et va remplacer la source de données configurée MySQL1.

Services JOnAS

Le contenu de ce guide est le suivant:

A qui il est destiné et pour quoi faire

Ce chapitre est destiné aux utilisateurs avancés de JOnAS qui ont besoin que quelques services "externes" tournent en coopeération avec le serveur JOnAS. Un service est une entité qui peut être initialisée, lancée et arrêtée. JOnAS lui−même définit déjà un ensemble de services, dont certains sont des pierres angulaires du serveur. Les services JOnAS prédéfinis sont listés dans Configurer les services JOnAS.

Les développeurs d'application J2EE peuvent avoir besoin d'accéder à d'autres services, par exemple un autre conteneur Web ou un conteneur Versant, pour leurs composants. Ainsi, il est important que de tels services puissent travailler de concert avec un serveur d'application. Pour atteindre cet objectif, il est possible de les définir en temps que services JOnAS.

Ce chapitre décrit comment définir un nouveau service JOnAS et comment spécifier quel service doit être lancé avec le serveur JOnAS.

Introduire un nouveau Service

La façon habituelle de définir un nouveau service JOnAS est de l'encapsuler dans une classe dont l'interface est connue de JOnAS. De façon plus précise, une telle classe fournit un moyen d'initialiser, de lancer et d'arrêter le service. Le fichier jonas.properties doit ensuite être modifié pour que JOnAS ait connaissance de ce service.

Définir la classe Service

Un service JOnAS est représenté par une classe qui implémente l'interface `org.objectweb.jonas.service.Service`, ainsi censée implémenter les méthodes suivantes :

- `public void init(Context ctx) throws ServiceException;`
- `public void start() throws ServiceException;`
- `public void stop() throws ServiceException;`
- `public boolean isStarted();`
- `public String getName();`
- `public void setName(String name);`

Il doit également définir un constructeur public sans argument.

Ces méthodes seront appelées par JOnAS pour initialiser, lancer, et arrêter le service. Les paramètres de configuration sont passés à la méthode d'initialisation au travers d'un contexte de nommage. Ce contexte de nommage est construit à partir des propriétés définies dans le fichier `jonas.properties` comme indiqué dans la section suivante.

La classe service devrait ressembler à ceci :

```
package a.b;
import javax.naming.Context;
import javax.naming.NamingException;
import org.objectweb.jonas.service.Service;
import org.objectweb.jonas.service.ServiceException;
.....
public class MyService implements Service {
    private String name = null;
    private boolean started = false;
    .....
    public void init(Context ctx) throws ServiceException {
        try {
            String p1 = (String) ctx.lookup("jonas.service.serv1.p1");
            .....
        } catch (NamingException e) {
            throw new ServiceException("....", e);
        }
        .....
    }
    public void start() throws ServiceException {
        .....
        this.started = true;
    }
    public void stop() throws ServiceException {
        if (this.started) {
            this.started = false;
            .....
        }
    }
    public boolean isStarted() {
        return this.started;
    }
    public String getName() {
        return this.name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

Modifier le fichier jonas.properties

Le service est défini et ses paramètres d'initialisation spécifiés dans le fichier `jonas.properties`. En premier, il faut choisir un nom pour le service (e.g. "serv1"), puis réaliser les opérations suivantes :

- ajouter ce nom à la propriété `jonas.services` ; cette propriété définit l'ensemble de services (séparés par des virgules) qui vont être lancés avec JOnAS, ***dans l'ordre de cette liste***.
- ajouter une propriété `jonas.service.serv1.class` spécifiant la classe service.
- ajouter autant de propriétés `jonas.service.serv1.XXX` spécifiant les paramètres d'initialisation du service, qu'il y en aura qui seront rendues accessibles à la classe service au moyen de l'argument Context de la méthode `init`.

Ceci est illustré ci−dessous :

```
jonas.services                  .......,serv1
jonas.service.serv1.class       a.b.MyService
jonas.service.serv1.p1          value
```

Utiliser le nouveau Service

Le nom donné au nouveau service se trouve dans `jonas.properties`. Avec ce nom, il est possible d'obtenir une référence sur la classe d'implémentation du service en utilisant la méthode de ServiceManager : `getService(name)`. Ci−dessous un exemple d'accès à un Service :

```
import org.objectweb.jonas.service.ServiceException;
import org.objectweb.jonas.service.ServiceManager;

    MyService sv = null;

        // Get a reference on MyService.
        try {
            sv = (MyService) ServiceManager.getInstance().getService("serv1");
        } catch (ServiceException e) {
            Trace.errln("Cannot find MyService:"+e);
        }
```

Ajouter la classe du nouveau service à JOnAS

Packager la classe du service dans un fichier .jar et ajouter le jar dans le répertoire `JONAS_ROOT/lib/ext`. Toutes les bibliothèques nécessitées par le service peuvent également être placées dans ce répertoire.

Compréhension avancée

Se référer aux sources JOnAS pour plus de détails sur les classes mentionnées dans cette section.

Services intégrés à JOnAS

Les services JOnAS existants sont les suivants :

Nom du Service	*Classe Service*
registry	RegistryServiceImpl
ejb	EJBServiceImpl
web	CatalinaJWebContainerServiceImpl / JettyJWebContainerServiceImpl
ear	EarServiceImpl
dbm	DataBaseServiceImpl
jms	JmsServiceImpl
jmx	JmxServiceImpl
jtm	TransactionServiceImpl
mail	MailServiceImpl
resource	ResourceServiceImpl
security	JonasSecurityServiceImpl
ws	AxisWSService

Si tous ces services sont nécessaires, ils seront lancés dans l'ordre suivant: *registry*, *jmx*, *security*, *jtm*,*dbm*,*mail*,*jms*,*resource*,*ejb*, ws, *web*, *ear*.
jmx, *security*, *dbm*, *mail*, *resource* sont optionnels quand vous utilisez le service *ejb*.

registry doit être lancé en premier.
(A noter que pour des raisons de compatibilité avec des versions antérieures de JOnAS, si *registry* est de façon involontaire non positionné comme le premier service à lancer, JOnAS va automatiquement lancer le service *registry*.)

A noter que *dbm*, *jms*, *resource*, et *ejb* dépendent de *jtm*.
A noter que *ear* dépend de *ejb* et *web* (qui fournissent les conteneurs ejb et web), c'est pourquoi ces services doivent être lancés avant le service *ear*.
A noter que ear et web dépendent de ws, c'est pourquoi le service ws doit être lancé avant les services *ear* et web.

Il est possible de lancer un Transaction Manager autonome en ne spécifiant que les services *registry* et *jtm*.

Un fichier `jonas.properties` ressemblera au suivant :

```
.....
.....

jonas.services                registry,jmx,security,jtm,dbm,mail,jms,ejb,resource,serv1

jonas.service.registry.class  org.objectweb.jonas.registry.RegistryServiceImpl
jonas.service.registry.mode        automatic

jonas.service.dbm.class       org.objectweb.jonas.dbm.DataBaseServiceImpl
jonas.service.dbm.datasources Oracle1

jonas.service.ejb.class       org.objectweb.jonas.container.EJBServiceImpl
```

```
jonas.service.ejb.descriptors ejb-jar.jar
jonas.service.ejb.parsingwithvalidation  true
jonas.service.ejb.mdbthreadpoolsize      10

jonas.service.web.class        org.objectweb.jonas.web.catalina.CatalinaJWebContainerServiceImpl
jonas.service.web.descriptors        war.war
jonas.service.web.parsingwithvalidation  true

jonas.service.ear.class       org.objectweb.jonas.ear.EarServiceImpl
jonas.service.ear.descriptors j2ee-application.ear
jonas.service.ear.parsingwithvalidation  true

jonas.service.jms.class       org.objectweb.jonas.jms.JmsServiceImpl
jonas.service.jms.mom         org.objectweb.jonas_jms.JmsAdminForJoram
jonas.service.jms.collocated  true
jonas.service.jms.url         joram://localhost:16010

jonas.service.jmx.class       org.objectweb.jonas.jmx.JmxServiceImpl

jonas.service.jtm.class       org.objectweb.jonas.jtm.TransactionServiceImpl
jonas.service.jtm.remote      false
jonas.service.jtm.timeout     60

jonas.service.mail.class        org.objectweb.jonas.mail.MailServiceImpl
jonas.service.mail.factories  MailSession1

jonas.service.security.class  org.objectweb.jonas.security.JonasSecurityServiceImpl

jonas.service.resource.class        org.objectweb.jonas.resource.ResourceServiceImpl
jonas.service.resource.resources         MyRA

jonas.service.serv1.class     a.b.MyService
jonas.service.serv1.p1        John
```

Le ServiceException

L'exception `org.objectweb.jonas.service.ServiceException` est définie pour les Services. Son type est `java.lang.RuntimeException.` et elle peut encapsuler n'importe quelle `java.lang.Throwable`.

Le ServiceManager

La classe `org.objectweb.jonas.service.ServiceManager` est responsable pour la création, l'initialisation, et le lancement des services. Il peut également renvoyer un service à partir de son nom et lister l'ensemble des services.

Guide de l'utilisateur JMS

Comme le requièrent les spécifications J2EE v1.3, les composants d'application (servlets, pages JSP, et bean d'entreprise) peuvent utiliser JMS pour la programmation par message Java. De plus, les applications peuvent utiliser les EJB orientés message (Message−driven Beans) pour des invocations asynchrones de méthodes EJB , comme spécifié par la spécification EJB 2.0. Depuis la version de JOnAS 3.1, JOnAS supporte la Spécification 1.1 du Java Message Service. Auparavant, dans le JMS 1.0.2, la programmation client pour le Point à point (un émetteur, un récepteur) et pour le Pub/Sub (Publish/Subscribe − multiples récepteurs) était réalisée en utilisant des hiérarchies de classes similaires, mais séparées. Maintenant, le JMS 1.1 offre une approche indépendante du domaine (Point à point ou pub/sub) pour la programmation de l'application client. Ainsi, le modèle de programmation est simplifié et il est maintenant possible d'engager des files d'attente (queues) et des sujets (topics) dans la même transaction.

Les fournisseurs d'EJB peuvent utiliser des ressources JMS Connection Factory (Fabrique de connexions) via des références de ressources, et des ressources JMS Destination (JMS Queues et JMS Topics) via des références d'environnement de ressources Ainsi, ils sont capables de fournir du code JMS, à l'intérieur d'une méthode EJB ou d'une méthode d'un composant web, pour envoyer ou recevoir de manière synchrone des messages vers/ ou à partir des Queues ou des Topics JMS . Le conteneur d'EJB et le conteneur web permettent d'effectuer des opérations JMS dans une transaction globale qui peut inclure d'autres ressources telles que des base de données.

JOnAS intègre une implémentation JMS tiers ; c'est actuellement **JORAM**, une implémentation open source, qui est intégrée et fournie avec JOnAS. De plus, les produits SwiftMQ, et IBM's WebSphere MQ peuvent être utilisés. D'autres implémentations JMS peuvent aussi être intégrées facilement. Les objets administrés par JMS qui sont utilisés par les composants de l'application, comme les "Connections Factories" et les "Destinations", peuvent être créés avant l'exécution via les services propriétaires d'administration de l'implémentation JMS. Toutefois, JOnAS fournit des « wrappers » sur des telles API d'administration JMS, permettant à JOnAS lui−même de réaliser automatiquement des opérations simples d'administration, et évitant au déployeur d'avoir à utiliser les APIs ou outils propriétaires du JMS embarqué. Se référer à Administration de JMS.

Installation et configuration de JMS

Pour utiliser JMS avec JOnAS, aucune opération d'installation ou de configuration n'est nécessaire. JOnAS contient :

- ·L'API 1.1 du Java[TM] Message Service , actuellement intégrée dans la distribution JOnAS
- ·Une implémentation JMS. L'OpenSource JORAM (http://joram.objectweb.org), **est intégrée dans la distribution JOnAS**, il n'y a donc pas d'installation à réaliser.

De plus, les produits SwiftMQ et IBM's WebSphere MQ sont utilisables avec JOnAS.

Ecriture d'opérations JMS au sein d'un composant d'application

Pour envoyer (ou recevoir de manière synchrone) des messages JMS, les composants requièrent l'accès à des objets administrés JMS, par exemple les Fabriques de Connexion pour créer des connexions aux ressources JMS, et des objets Destination (Queue ou Topics) qui sont des entités JMS utilisées comme destinations dans les opérations d'envoi JMS. Les deux sont rendus accessibles au travers du JNDI par le service d'administration du fournisseur de JMS.

Référerez−vous à l'exemple **`jms`** en complément de ce document. Cet exemple jms est décrit ici.

Accès à la Fabrique de Connexion (Connection Factory)

La spécification EJB introduit le concept de « Resource Manager Connection Factory References ». Ce concept apparaît aussi dans les spécifications J2EE v1.3. Il est utilisé pour créer des connexions au ressource manager. A ce jour, on considère 3 types de Ressource Manager Connection Factories :

- Les objects DataSource (javax.sql.DataSource) représentent les fabriques d'objets connexions JDBC.
- Les fabriques de connexions JMS (`javax.jms.ConnectionFactory`, `javax.jms.QueueConnectionFactory` et `javax.jms.TopicConnectionFactory`) sont les fabriques d'objets connexions JMS.
- Les fabriques de connexions Java Mail (`javax.mail.Session` ou `javax.mail.internet.MimePartDataSource`) sont les fabriques d'objets connexions Java Mail.

Les fabriques de connexion qui nous intéressent ici sont celles du deuxième type, les fabriques de connexion JMS.

Notez qu'à partir de JMS 1.1, il est recommandé de n'utiliser que les `javax.jms.ConnectionFactory` (plutôt que `javax.jms.QueueConnectionFactory` ou `javax.jms.TopicConnectionFactory`). Toutefois, la nouvelle implémentation est totalement compatible avec les versions antérieures, et les applications existantes pourront fonctionner telles quelles.

Le descripteur de déploiement standard doit contenir l'élément suivant `resource−ref` :

```
<resource−ref>
<res−ref−name>jms/conFact</res−ref−name>
<res−type>javax.jms.ConnectionFactory</res−type>
<res−auth>Container</res−auth>
</resource−ref>
```

Cela signifie que le développeur aura accès à l'objet ConnectionFactory en utilisant le nom JNDI *java:comp/env/jms/conFact*. Le code source pour obtenir l'objet factory est le suivant :

```
ConnectionFactory qcf = (ConnectionFactory)
           ctx.lookup("java:comp/env/jms/conFact");
```

Le mapping sur le véritable nom JNDI de la ConnectionFactory (comme défini avec l'outil d'administration JMS du

fournisseur), CF dans l'exemple, est défini dans le descripteur de déploiement spécifique à JOnAS avec l'élément suivant :

```
<jonas-resource>
<res-ref-name>jms/conFact</res-ref-name>
<jndi-name>CF</jndi-name>
</jonas-resource>
```

Accès à l'objet Destination

Accéder à une destination JMS dans le code d'un composant d'application requière d'utiliser une *Resource Environment Reference*, qui est représentée dans le fichier standard de déploiement par :

```
<resource-env-ref>
<resource-env-ref-name>jms/stockQueue</resource-env-ref-name>
<resource-env-ref-type>javax.jms.Queue<resource-env-ref-type>
</resource-env-ref>
```

Le code source du composant de l'application doit contenir :

```
Queue q = (Queue) ctx.lookup("java:comp/env/jms/stockQueue");
```

Le mapping sur le véritable nom JNDI (ex : « myQueue ») étant défini dans le descripteur de déploiement spécifique de JOnAS de la façon suivante :

```
<jonas-resource-env>
<resource-env-ref-name>jms/stockQueue</resource-env-ref-name>
<jndi-name>myQueue<jndi-name>
</jonas-resource-env>
```

Ecriture d'opérations JMS

Une méthode typique pour réaliser l'envoi d'un message JMS ressemblera à ceci :

```
void sendMyMessage() {

  ConnectionFactory cf = (ConnectionFactory)
          ctx.lookup("java:comp/env/jms/conFact");
  Queue queue = (Queue) ctx.lookup("java:comp/env/jms/stockQueue");
  Connection conn = cf.createConnection();
  Session sess = conn.createSession(true, Session.AUTO_ACKNOWLEDGE);



  MessageProducer  mp = sess.createProducer((Destination)queue);
  ObjectMessage msg = sess.createObjectMessage();
  msg.setObject("Hello");
  sender.send(msg);
  sess.close();
```

```
    conn.close();
  }
```

Un composant d'application peut aussi recevoir de manière synchrone un message. Une méthode EJB permettant de réaliser la réception synchrone d'un message d'une liste d'attente est illustrée ici :

```
public String recMsg() {
    ConnectionFactory cf = (ConnectionFactory)
           ctx.lookup("java:comp/env/jms/conFact");
    Queue queue = (Queue) ctx.lookup("java:comp/env/jms/stockQueue");
    Connection conn = cf.createConnection();
    Session sess = conn.createSession(true, Session.AUTO_ACKNOWLEDGE);
    MessageConsumer mc = sess.createConsumer((Destination)queue);
    conn.start();
    ObjectMessage msg = (ObjectMessage) mc.receive();
    String msgtxt =  (String) msg.getObject();
    sess.close();
    conn.close();
    return msgtxt;
}
```

Une méthode qui effectue des opérations JMS doit toujours contenir les clauses de création et de fermeture de session, comme suit :

```
 public void doSomethingWithJMS (...) {
   ...
   session = connection.createSession(...);
   ... // JMS operations
   session.close();
 }
```

Les opérations JMS feront partie de la transaction, si il y en a une, quand le serveur JOnAS exécute la méthode.

Assurez−vous de ne jamais envoyer ***et*** recevoir un message particulier dans la même transaction, car les opérations d'envoi JMS ne sont effectivement réalisées qu'au moment de l'exécution du "commit" de la transaction.

Les exemples précédents illustraient la messagerie point à point. Toutefois, les composants d'application peuvent aussi être développés en utilisant l'API JMS publish/subscribe, par exemple en utilisant le type de destination `Topic` au lieu de `Queue`. This offers the capability of broadcasting a message to several message consumers at the same time. Cela offre la possibilité de diffuser un message à différents clients en même temps. L'exemple suivant illustre une méthode pour publier un message sur un topic JMS et démontre comment les interfaces ont été simplifiées depuis JMS 1.1.

```
public void sendMsg(java.lang.String s) {
    ConnectionFactory cf = (ConnectionFactory)
                   ictx.lookup("java:comp/env/jms/conFactSender");
    Topic topic = (Topic) ictx.lookup("java:comp/env/jms/topiclistener");
    Connection conn = cf.createConnection();
    Session session = conn.createSession(true, Session.AUTO_ACKNOWLEDGE);
    MessageConsumer mc = session.createConsumer((Destination)topic);
```

```
        ObjectMessage message = session.createObjectMessage();
        message.setObject(s);
        mc.send(message);
        session.close();
        conn.close();
    }
```

Transactions et sessions JMS dans un composant d'application

La création de session JMS au sein d'un composant d'application, peut donner lieu à différents comportements, suivant si la session est créée à l'exécution à l'intérieur ou à l'extérieur d'une transaction. En fait, les paramètres de la méthode `createSession(boolean transacted, int acknowledgeMode)` *ne sont jamais pris en compte*.

- Si la création de la session s'effectue en dehors de toute transaction, les paramètres sont considérés comme étant `transacted = false` et `acknowledgeMode = AUTO_ACKNOWLEDGE`. Cela signifie que chaque opération de la session est exécutée immédiatement.
- Si la création de la session s'effectue à l'intérieur d'une transaction, les paramètres n'ont aucune signification, la session est considérée comme transactionnelle, et les opérations de validation (commit) et d'annulation (rollback) sont gérées par JOnAS, au niveau de la ressource XA associée.

Authentification

Si votre implémentation JMS permet l'authentification d'utilisateur, les méthodes suivantes peuvent être utilisées sur les Connection Factories :

- `createConnection(String userName, String password)` on `ConnectionFactory`
- `createQueueConnection(String userName, String password)` on `QueueConnectionFactory`
- `createTopicConnection(String userName, String password)` on `TopicConnectionFactory`

Quelques règles de programmation et restrictions quand on utilise JMS dans les EJB

Cette section présente quelques restrictions et règles de programmation lors de l'utilisation d'opérations JMS au sein de composants EJB.

Gestion de connexions

Suivant l'implémentation JMS et l'application, il peut être souhaitable de conserver les connexions JMS ouvertes pour la durée de vie de l'instance du bean, ou pour la durée de l'appel de méthode. Ces deux modes de programmation sont illustrés dans les exemples suivants (cet exemple illustre un bean session avec état) :

```
public class EjbCompBean implements SessionBean {
    ...
```

```
    QueueConnectionFactory qcf = null;
    Queue queue = null;

    public void ejbCreate() {
       ....
       ictx = new InitialContext();
       qcf = (QueueConnectionFactory)
          ictx.lookup("java:comp/env/jms/conFactSender");
       queue = (Queue) ictx.lookup("java:comp/env/jms/queue1");
    }

    public void doSomethingWithJMS (...) {
       ...
       Connection conn = qcf.createConnection();
       Session session = conn.createSession(...);
       ... // JMS operations
       session.close();
       conn.close();
     }

    ...
}
```

Pour garder la connexion ouverte durant la vie de l'instance du bean, le style de programmation de l'exemple suivant est préférable, puisqu'il évite de nombreuses opérations d'ouverture et de fermeture de connexion :

```
public class EjbCompBean implements SessionBean {
    ...
    ConnectionFactory qcf = null;
    Queue queue = null;
    Connection conn = null;

    public void ejbCreate() {
       ....
       ictx = new InitialContext();
       cf = (ConnectionFactory)
          ictx.lookup("java:comp/env/jms/conFactSender");
       queue = (Queue) ictx.lookup("queue1");
       conn = cf.createConnection();
    }

    public void doSomethingWithJMS (...) {
       ...
       Session session = conn.createSession(...);
       ... // JMS operations
       session.close();
    }

    public void ejbRemove() {
       conn.close();
    }

    ...
```

```
}
```

Soyez conscients que de maintenir des objets JMS dans l'état du bean n'est pas toujours possible, cela dépend du type de bean :

- ·Pour un bean session sans état (stateless session bean), l'état du bean n'est pas maintenu au travers des appels de méthodes. Par conséquent, les objets JMS doivent toujours être initialisés et définis dans chaque méthode réalisant des opérations JMS.
- ·Pour un bean entité (entity bean), une instance peut être « passivée », et seule la partie persistante de l'état du bean est maintenue. Il est donc recommandé que les objets JMS soient initialisés et définis dans chaque méthode réalisant des opérations JMS. Si ces objets sont définis dans l'état du bean, ils peuvent être initialisés dans la méthode `ejbActivate` (si la connexion est créée dans la méthode `ejbActivate`, vérifiez bien de la fermer dans la méthode `ejbPassivate)`.
- ·Pour un bean session avec état (stateful session bean), comme cela a été montré dans l'exemple précédent, les objets JMS peuvent être définis dans l'état du bean. Les instances des beans sessions avec état peuvent être « passivées » (pas dans la version actuelle de JOnAS). Comme les fabriques de connexions et les destinations sont des objets sérialisables, ils peuvent être initialisés dans un `ejbCreate`. Toutefois, faites attention qu'une connexion doit être fermée dans un `ejbPassivate` (avec la variable d'état fixée à null) et recréée dans `ejbActivate`.

A noter : à cause d'un problème identifié avec le JDK 1.3 de Sun sur Linux, la fermeture des connexions peut bloquer. Ce problème a été résolu avec le JKD 1.4.

Démarrer des transactions après la création de connexion ou de session JMS

Actuellement il n'est pas possible de démarrer une transaction gérée par un bean après la création d'une session JMS, et d'avoir les opérations JMS impliquées dans la transaction. Dans l'exemple de code suivant, les opérations JMS ne s'effectueront pas dans la transaction `ut` :

```
public class EjbCompBean implements SessionBean {
    ...

    public void doSomethingWithJMS (...) {
       ...
       Connection conn = cf.createConnection();
       Session session = conn.createSession(...);
       ut = ejbContext.getUserTransaction();
       ut.begin();
       ... // JMS operations
       ut.commit();
       session.close();
       conn.close();
     }

    ...
}
```

Pour que les opérations JMS de la session soient impliquées dans la transaction, il faut que les opérations de création et de fermeture de la session se trouvent à l'intérieur des limites de la transaction ; les opérations de création et de fermeture de la connexion peuvent, elles, être soit à l'extérieur soit à l'intérieur des limites de la transaction, comme suit :

```
public class EjbCompBean implements SessionBean {
    ...

    public void doSomethingWithJMS (...) {
       ...
       Connection conn = qcf.createConnection();
       ut = ejbContext.getUserTransaction();
       ut.begin();
       Session session = conn.createSession(...);
       ... // JMS operations
       session.close();
       ut.commit();
       conn.close();
     }

    ...
}
```

ou

```
public class EjbCompBean implements SessionBean {
    ...

    public void doSomethingWithJMS (...) {
       ...
       ut = ejbContext.getUserTransaction();
       ut.begin();
       Connection conn = cf.createConnection();
       Session session = conn.createSession(...);
       ... // JMS operations
       session.close();
       conn.close();
       ut.commit();
     }

    ...
}
```

Programmer des composants EJB avec des transactions gérées par les beans peut rendre le code complexe. L'utilisation de transactions gérées par le conteneur peut permettre d'éviter des problèmes comme ceux que nous venons de voir.

Administration JMS

Les applications utilisant la communication par message requièrent des objets administrés par JMS : les *fabriques de connexions* et les *destinations*. Ces objets sont créés via une interface d'administration propriétaire (non standardisée) du fournisseur JMS. Pour les cas simples d'administration et pour les implémentations intégrées à JOnAS (comme JORAM ou SwiftMQ), cette phase d'administration peut être évitée, en laissant JOnAS réaliser ces fonctions, comme cela est décrit dans la section suivante "Administration JMS simplifiée avec JOnAS" section.

Administration JMS simplifiée avec JOnAS

Dans la configuration par défaut, JOnAS crée automatiquement six objets "fabriques de connexions" et deux "destinations". Pour utiliser JMS dans cette configuration, la seule chose nécessaire est de configurer l'utilisation du service JMS dans le fichier ***jonas.properties***.

```
jonas.services           security,jtm,dbm,jms,ejb
```

Les 6 "fabriques de connexions" créées automatiquement sont décrites dans le tableau suivant :

Nom JNDI	Type JMS	Utilisation
CF	ConnectionFactory	Utilisé par un composant d'application pour créer une Connection.
QCF	QueueConnectionFactory	Utilisé par un composant d'application pour créer une QueueConnection.
TCF	TopicConnectionFactory	Utilisé par un composant d'application pour créer une TopicConnection.
JCF	ConnectionFactory	Utilisé par tout autre composant Java (par exemple un client) pour créer une Connection.
JQCF	QueueConnectionFactory	Utilisé par tout autre composant Java (par exemple un client) pour créer une QueueConnection.
JTCF	TopicConnectionFactory	Utilisé par tout autre composant Java (par exemple un client) pour créer une TopicConnection.

Les fabriques de connexions CF, QCF et TCF sont des "fabriques de connexions *gérées*". Les composants d'application ne devraient utiliser que des "fabriques de connexions gérées" pour permettre à JOnAS de gérer les ressources JMS créées via ces fabriques de connexions (les sessions JMS).

A l'inverse, JCF, JQCF et JTCF ne sont pas gérées. Elles sont utilisées par des composants Java implémentant le comportement d'un client JMS, mais tournant hors du serveur d'application.

Avec le fichier **jonas.properties** par défaut, deux objets destinations sont créés lorsque le service JMS est lancé :

JNDI name	JMS type	Usage
sampleQueue	Queue	Peut être utilisé indifféremment par un composant EJB ou tout autre composant Java
sampleTopic	Topic	Peut être utilisé indifféremment par un composant EJB ou tout autre composant Java

JOnAS ne créera pas d'autres fabriques de connexions avec sa configuration par défaut. Cependant il peut créer des objets destination requis au lancement du serveur, si cela est spécifié dans le fichier **jonas.properties**. Pour cela, il faut spécifier les noms JNDI des destinations Topic et Queue à créer dans les propriétés respectives **jonas.service.jms.topics** et **jonas.service.jms.queues**, comme suit :

```
jonas.service.jms.topics    t1,t2    // JOnAS server creates 2 topic destinations (t1,t2)
jonas.service.jms.queues    myQueue  // JOnAS server creates 1 queue destination (myQueue)
```

Ceci peut aussi être fait dynamiquement via la console d'administration de JOnAS.

Il est recommandé que les développeurs utilisent des *références de ressources*, et des *références d'environnement de ressources* pour accéder aux fabriques de connexions et aux objets destinations créés par JOnAS, comme cela a déjà été présenté dans la section "Ecriture d'opérations JMS au sein d'un composant d'application".

Exécution d'un EJB réalisant des opérations JMS

Tout ce qu'il est nécessaire de faire est :

```
jonas start
```

Le MOM (Message−Oriented Middleware), qui est l'implémentation du fournisseur de JMS, est démarré automatiquement, et les objets administrés JMS qui vont être utilisés par les EJB sont automatiquement créés et enregistrés dans le JNDI. Ensuite les EJB peuvent être déployés comme d'habitude avec :

```
jonas admin −a XX.jar
```

Lancement du Message−Oriented Middleware

Si la propriété JOnAS **jonas.services** contient le service **jms**, celui−ci sera démarré et tentera de lancer une implémentation de JMS (e.g. le MOM JORAM ou le MOM SwiftMQ).

Pour lancer le MOM, les possibilités suivantes sont offertes :

1. **Lancement automatique du MOM dans la JVM de JOnAS**
   Ceci est réalisé en utilisant les valeurs par défaut pour les options de configuration, par exemple en laissant la valeur ***jonas.service.jms.collocated*** a `true` dans le fichier ***jonas.properties*** (voir le fichier ***jonas.properties*** fourni dans le répertoire `$JONAS_ROOT/conf`).

   ```
   jonas.service.jms.collocated true
   ```

   Dans ce cas, le MOM sera lancé automatiquement au démarrage du serveur (commande `jonas start`).

Note pour utiliser le MOM JORAM depuis un hôte distant :

- Pour utiliser les ressources JMS à partir d'un hôte distant, la valeur de la propriété ***hostname*** dans le fichier de configuration par défaut ***a3servers.xml*** doit être changée, passant de la valeur ***localhost*** au nom de l'hôte distant. Voir le cas 4 ci−dessous (lancement du MOM sur un numéro de port particulier) pour les détails de configuration de JORAM.

2. **Lancement du MOM dans un JVM séparée sur le même hôte**
   Le MOM JORAM peut−être lancé avec les options par défaut en utilisant la commande :
   `JmsServer`

   Pour d'autres MOM, utilisez les commandes propriétaires.

   Dans ce cas la propriété JOnAS **jonas.service.jms.collocated** doit être positionnée à `false` dans le fichier **jonas.properties**.

   ```
   jonas.service.jms.collocated false
   ```

3. **Lancement du MOM sur un hôte distinct**

   Le MOM peut−être lancé sur un hôte séparé. Dans ce cas, le serveur JOnAS doit être informé que le MOM tourne sur un autre hôte via la propriété de JOnAS **jonas.service.jms.url** dans le fichier **jonas.properties**. Pour JORAM, sa valeur doit alors être l'URL JORAM `joram://host:port` où `host` est le nom du hôte, et `port` le numéro de port par défaut de JORAM, i.e. 16010 (Pour SwiftMQ, la valeur de l'URL est similaire à `smqp://host:4001/timeout=10000`).

   ```
   jonas.service.jms.collocated false
   jonas.service.jms.url       joram://host2:16010
   ```

4. **Lancement du MOM sur un numéro de port particulier (pour JORAM)**

   Pour changer le numéro de port par défaut de JORAM, il faut faire une opération de configuration spécifique à JORAM (en modifiant le fichier de configuration ***a3servers.xml*** qui se trouve dans le répertoire où JORAM est explicitement lancé). Un fichier `a3servers.xml` est fourni dans le répertoire $JONAS_ROOT/conf. Ce fichier spécifie que le MOM tourne sur le localhost et utilise le numéro de port par défaut de JORAM.

   Pour lancer le MOM sur un autre numéro de port, il faut modifier l'attribut args de l'élément classe de service "fr.dyade.aaa.mom.ConnectionFactory" dans le fichier `a3servers.xml`, et mettre à jour la propriété **jonas.service.jms.url** dans le fichier **jonas.properties**.

   Le fichier par défaut a3servers.xml est situé dans le répertoire $JONAS_ROOT/conf. Pour utiliser un fichier de configuration situé dans un endroit différent, il faut passer la propriété système `−Dfr.dyade.aaa.agent.A3CONF_DIR="votre répertoire pour a3servers.xml"` au

lancement du MOM.
Pour modifier d'autres éléments de la configuration du MOM (distribution, multi−serveurs, ...),référez−vous à la documentation JORAM à http://joram.objectweb.org.

Note: le MOM peut être lancé avec sa commande propriétaire. Pour JORAM la commande est :

```
java −DTransaction=NullTransaction fr.dyade.aaa.agent.AgentServer 0 ./s0
```

Cette commande correspond aux options par défaut prises par la commande `JmsServer`.

Les messages JMS ne sont pas persistants quand on lance le MOM avec cette commande. Si des messages persistants sont requis, l'option `−DTransaction=NullTransaction` doit être remplacée par l'option `−DTransaction=Atransaction`. Veuillez vous référer à la documentation de JORAM pour plus de détails sur cette commande.

Un exemple d'EJB JMS

Cet exemple montre une application à base d'EJB qui combine un bean d' entreprise qui envoie un message JMS, et un bean d'entreprise qui écrit dans une base de données (un bean entité) dans le même transaction globale. Il est composé des éléments suivants :

- Un bean session, EjbComp, avec une méthode qui envoie un message sur un topic JMS.
- ·Un bean entité, Account (celui utilisé dans l'exemple eb, avec une persistance gérée par le conteneur) qui écrit sa donnée dans une table de base de données relationnelle, et représente un message envoyé (c'est−à−dire, chaque fois qu'un bean EjbComp envoie un message, une instance du bean entité sera créée).
- Un EJB client, EjbCompClient, qui appelle la méthode sendMsg du bean EjbComp et crée un bean entité Account, le tout dans la même transaction. Lors de la validation d'une telle transaction, le message JMS est effectivement envoyé et l'enregistrement correspondant à l'instance du bean entité dans la base de données est créé. Lors d'un "rollback", le message n'est pas envoyé, et rien n'est créé dans la base de données.
- Un pure client JMS MsgReceptor, tournant en dehors du serveur JOnAS, et dont le rôle est de recevoir les messages envoyés par le bean d'entreprise sur le topic.

Le bean sessionqui effectue les opérations JMS

Le bean doit contenir le code pour initialiser les références aux objets administrés JMS qu'il utilisera. Pour éviter la répétition de ce code dans chaque méthode qui réalise une opération JMS, on peut l'introduire dans la méthode `ejbCreate`.

```
public class EjbCompBean implements SessionBean {
    ...
    ConnectionFactory cf = null;
    Topic topic = null;

    public void ejbCreate() {
       ....
       ictx = new InitialContext();
```

```
        cf = (ConnectionFactory)
           ictx.lookup("java:comp/env/jms/conFactSender");
        topic = (Topic) ictx.lookup("java:comp/env/jms/topiclistener");
    }
    ...
}
```

Ce code a été intentionnellement nettoyé de tous les éléments qui ne sont pas nécessaires à la compréhension des aspects logiques JMS de l'exemple, comme par exemple la gestion des exceptions.

Les objets JMS administrés `ConnectionFactory` et `Topic` sont rendus disponibles au bean par une *resource reference* dans le premier cas, et par une *resource environment reference* dans le second cas.
Le descripteur de déploiement standard doit contenir l'élément suivant :

```
      <resource-ref>
        <res-ref-name>jms/conFactSender</res-ref-name>
        <res-type>javax.jms.ConnectionFactory</res-type>
        <res-auth>Container</res-auth>
      </resource-ref>

      <resource-env-ref>
        <resource-env-ref-name>jms/topiclistener</resource-env-ref-name>
        <resource-env-ref-type>javax.jms.Topic</resource-env-ref-type>
      </resource-env-ref>
```

Le descripteur de déploiement spécifique JOnAS doit contenir l'élément suivant :

```
      <jonas-resource>
        <res-ref-name>jms/conFactSender</res-ref-name>
        <jndi-name>TCF</jndi-name>
      </jonas-resource>

      <jonas-resource-env>
        <resource-env-ref-name>jms/topiclistener</resource-env-ref-name>
        <jndi-name>sampleTopic</jndi-name>
      </jonas-resource-env>
```

A noter que le bean session EjbComp va utiliser les objets administrés automatiquement créés par JOnAS dans sa configuration par défaut.

Une fois que les objets administrés sont accessibles, il est possible d'effectuer des opérations JMS à l'intérieur des méthodes. C'est ce qui se passe dans la méthode `sendMsg` :

```
public class EjbCompBean implements SessionBean {
    ...
    public void sendMsg(java.lang.String s) {
        // create Connection, Session and MessageProducer
        Connection conn = null;
        Session session = null;
        MessageProducer mp = null;
```

```
        try {
            conn = cf.createConnection();
            session = conn.createSession(true, Session.AUTO_ACKNOWLEDGE);
            mp = session.createProducer((Destination)topic);
        }
        catch (Exception e) {e.printStackTrace();}

        // send the message to the topic
        try {
            ObjectMessage message;
            message = session.createObjectMessage();
            message.setObject(s);
            mp.send(message);
            session.close();
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    ...
}
```

Cette méthode envoie un message contenant un argument String.

Le bean entité

L'exemple utilise le bean entité simple Account pour écrire des données dans la base de données. Se référer à l'exemple eb.

L'application cliente

L'application cliente appelle la méthode sendMsg du bean EjbComp et crée un bean entité AccountImpl, le tout dans la même transaction.

```
public class EjbCompClient {
    ...
    public static void main(String[] arg) {
    ...
    utx = (UserTransaction) initialContext.lookup("javax.transaction.UserTransaction");
    ...
    home1 = (EjbCompHome) initialContext.lookup("EjbCompHome");
    home2 = (AccountHome) initialContext.lookup("AccountImplHome");
    ...
    EjbComp aJmsBean = home1.create();
    Account aDataBean = null;
    ...
    utx.begin();
    aJmsBean.sendMsg("Hello commit"); // sending a JMS message
    aDataBean = home2.create(222, "JMS Sample OK", 0);
    utx.commit();
```

```
    utx.begin();
    aJmsBean.sendMsg("Hello rollback"); // sending a JMS message
    aDataBean = home2.create(223, "JMS Sample KO", 0);
    utx.rollback();
    ...
    }
}
```

Le résultat de l'exécution de l'aplication cliente est le suivant :

- le message `"Hello commit"` est envoyé et l'enregistrement `[222, 'JMS Sample OK', 0]` est créé dans la base de données (correspondant à la création du bean entité `109`).
- le message `"Hello rollback"` ne sera jamais envoyé et l'enregistrement `[223, 'JMS Sample KO', 0]` ne sera pas créé dans la base de données(car la création du bean entité 110 va être annulée).

Un pure client JMS pour recevoir les messages

Dans cet exemple, les messages envoyés par le composant EJB sont reçus par un client JMS qui s'exécute en dehors du serveur JOnAS, mais écoute les messages sur le topic JMS "sampleTopic". Il utilise la ConnexionFactory "JCF" créée automatiquement par JOnAS.

```
public class MsgReceptor {

    static Context ictx = null;
    static ConnectionFactory cf = null;
    static Topic topic = null;

    public static void main(String[] arg) {

        ictx = new InitialContext();
        cf = (ConnectionFactory) ictx.lookup("JCF");
        topic = (Topic) ictx.lookup("sampleTopic");
        ...
        Connection conn = cf.createConnection();
        Session session =
               conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
        MessageConsumer mc = session.createConsumer((Destination)topic);

        MyListenerSimple listener = new MyListenerSimple();
            mc.setMessageListener(listener);
        conn.start();

        System.in.read(); // waiting for messages

        session.close();
        conn.close();
        ...
    }
}

public MyListenerSimple implements javax.jms.MessageListener {
```

```
   MyListenerSimple() {}

   public void onMessage(javax.jms.Message msg) {
      try {
      if(msg==null)
        System.out.println("Message: message null ");
      else {
        if(msg instanceof ObjectMessage) {
            String m = (String) ((ObjectMessage)msg).getObject();
            System.out.println ("JMS client: received message ======> " + m);
        } else if(msg instanceof TextMessage) {
            String m = ((TextMessage)msg).getText();
            System.out.println ("JMS client: received message ======> " + m);
        }
      }catch(Exception exc) {
          System.out.println("Exception caught :" + exc);
          exc.printStackTrace();
      }
   }
}
```

L'administration JMS

L'utilisation des valeurs par défaut des propriétés du fichier ***jonas.properties***, offre des facilités simplifiées pour l'administration de JMS, suffisantes pour cet exemple. Il n'y a que l'utilisation du service JMS qui doit être spécifiée :

```
    jonas.services               security,jtm,dbm,jms,ejb
```

Le serveur JMS est lancé au sein du serveur JOnAS, et un topic nommé sampleTopic est créé quand le serveur JOnAS est lancé par la commande "jonas start".

Manuel Utilisateur de la Tâche Ant EJB

Nouvel élément JOnAS (Java Open Application Server) pour la version actuelle de JOnAS

L'élément encapsulé `<jonas>` utilise l'outil spécifique `GenIC` pour fabriquer les stubs et skeletons spécifques à JOnAS et construire un fichier JAR qui puisse être déployé sur le Serveur d'Application JOnAS. Le processus de fabrication vérifie toujours si les stubs/skeletons EJB et le fichier EJB−JAR sont à jour, et va ainsi réaliser le travail minimum requis.

Une convention de nommage pour les descripteurs EJB est la plupart du temps utilisée pour préciser le nom du fichier JAR terminé. Par exemple, si le descripteur EJB `ejb/Account−ejb−jar.xml` est situé dans le répertoire du descripteur, l'élément `<jonas>` va chercher un fichier descripteur EJB spécifique à JOnAS appelé `ejb/Account−jonas−ejb−jar.xml`, et un fichier JAR appelé `ejb/Account.jar` va être écrit dans le répertoire destination. L'élément `<jonas>` peut également utiliser la convention de nommage JOnAS. En utilisant le même exemple que précédemment, le descripteur EJB peut également être appelé `ejb/Account.xml` ("pas de terminaison de nom de base ici") dans le répertoire du descripteur. L'élément `<jonas>` va ensuite chercher un fichier descripteur EJB spécifique à JOnAS appelé `ejb/jonas−Account.xml`. Cette convention ne suit pas exactement la recommandation de convention de nommage ejb−jar, mais elle est supportée pour assurer une compatibilité avec des versions antérieures de JOnAS.

A noter que lorsque les descripteurs EJB sont rajoutés au fichier JAR, ils sont automatiquement renommés `META−INF/ejb−jar.xml` et `META−INF/jonas−ejb−jar.xml`.

De plus, ce comportement de nommage peut être modifié en précisant les attributs dans la tâche ejbjar (par exemple, basejarname, basenameterminator, et flatdestdir) aussi bien que l'élément iplanet (par exemple, suffix). Se référer à la documentation appropriée pour davantage de détails.

Paramètres:

Attribut	Description	Nécessaire
destdir	Le répertoire de base dans lequel seront écrits les fichiers JAR générés. Chaque fichier JAR est écrit dans les répertoires qui correspondent à leur localisation au sein du namespace "`descriptordir`".	Oui
jonasroot	Le répertoire racine pour JOnAS.	Oui
jonasbase	Le répertoire de base pour JOnAS. Si omis, sera `jonasroot par défaut.`	Non
classpath	Le classpath utilisé à la génération des stubs et skeletons EJB. Si omis, le classpath spécifié dans la tâche parente "ejbjar" sera utilisée. S'il est spécifié, les éléments classpath seront préfixés au classpath spécifié dans la tâche parente "ejbjar". Un élément "classpath" encapsulé peut également être utilisé. A noter que les fichiers JAR JOnAS nécessaires sont rajoutés automatiquement au classpath.	Non

keepgenerated	`true` si les fichiers sources Java intermédiaires générés par GenIC doivent être conservés. Si omis, mis par défaut à `false`.	Non
nocompil	`true` si les fichiers sources générés ne doivent pas être compilés avec le compilateur java et rmi. Si omis, mis par défaut à `false`.	Non
novalidation	`true` si les descripteurs de déploiement XML doivent être parsés sans validation. Si omis, mis par défaut à `false`.	Non
javac	compilateur Java à utiliser. Si omis, mis par défaut à la propriété de la valeur `build.compiler`.	Non
javacopts	Options à passer au compilateur java.	Non
protocols	liste des protocoles séparés par une virgule (choisi parmi jeremie, jrmp, iiop, cmi) pour lesquels des stubs devraient être générés. Par défaut, `jrmp,jeremie`.	Non
rmicopts	Options à passer au compilateur rmi.	Non
verbose	Indique s'il faut ou non utiliser le switch −verbose. Si omis, mis par défaut à `false`.	Non
additionalargs	Rajoute des arguments (args) à GenIC.	Non
keepgeneric	`true` si le fichier JAR générique utilisé en entrée par GenIC doit être retenu. Si omis, mis par défaut à `false`.	Non
suffix	Valeur de la chaîne de caractères ajoutée en fin de nom de fichier JAR à la création de chaque JAR. Si omis, mis par défaut à ".jar".	Non
nogenic	Si cet attribut est mis à `true`, GenIC de JOnAS ne s'exécutera pas sur le JAR EJB. A utiliser si vous préférez éxécuter GenIC au moment du déploiement. Si omis, mis par défaut à `false`.	Non
mappernames	Liste des noms d'adapteurs JORM, séparés par une virgule, utilisé pour CMP2.0 pour indiquer pour quels adapteurs les classes conteneur devraient être générées.	Non
jvmopts	args (arguments) additionnels à passer à la JVM GenIC.	Non

Comme indiqué ci−dessus, l'élément jonas supporte des éléments encapsulés additionnels <classpath> .

Exemples

Cet exemple montre un ejbjar utilisé pour générer des jars de déploiement utilisant un conteneur EJB JOnAS. Cet exemple montre l'utilisation du nommage standard pour les descripteurs de déploiement. L'utilisation de ce format crée un fichier JAR EJB pour chaque variation de '*−jar.xml' qui est situé dans le répertoire du descripteur de déploiement.

```
<ejbjar srcdir="${build.classes}"
        descriptordir="${descriptor.dir}">
  <jonas destdir="${deploymentjars.dir}"
         jonasroot="${jonas.root}"
         protocols="jrmp,iiop"/>
  <include name="**/*.xml"/>
  <exclude name="**/jonas−*.xml"/>
  <support dir="${build.classes}">
```

```
        <include name="**/*.class"/>
  </support>
</ejbjar>
```

Cet exemple montre un ejbjar utilisé pour générer un simple jar de déploiement utilisant un conteneur EJB JOnAS. Cet exemple nécessite que les descripteurs de déploiement utilisent le nommage standard. Ceci crée un unique fichier jar ejb − 'TheEJBJar.jar'.

```
<ejbjar srcdir="${build.classes}"
        descriptordir="${descriptor.dir}"
        basejarname="TheEJBJar">
  <jonas destdir="${deploymentjars.dir}"
         jonasroot="${jonas.root}"
         suffix=".jar"
         protocols="${genic.protocols}"/>
  <include name="**/ejb-jar.xml"/>
  <exclude name="**/jonas-ejb-jar.xml"/>
</ejbjar>
```

Guide d'utilisation des Modules de Login dans un client Java

Le contenu de ce guide est le suivant :

Configurer un environnement pour utilier des modules de login avec des clients java

Les modules de login à utiliser par les clients sont définis dans le fichier `$JONAS_ROOT/conf/jaas.config`. Exemple :

```
jaasclient {
    // Login Module to use for the example jaasclient.

    //First, use a LoginModule for the authentication
    // Use the resource memrlm_1
    org.objectweb.jonas.security.auth.spi.JResourceLoginModule required
    resourceName="memrlm_1"
        ;

    // Use the login module to propagate security to the JOnAS server
    // globalCtx is set to true in order to set the security context
    // for all the threads of the client container instead of only
    // on the current thread.
    // Useful with multithread applications (like Swing Clients)
    org.objectweb.jonas.security.auth.spi.ClientLoginModule  required
    globalCtx="true"
        ;
};
```

Ce fichier est utilisé quand un client java est lancé avec `jclient`, car ce script fixe la propriété suivante : `-Djava.security.auth.login.config==$JONAS_ROOT/conf/jaas.config`

Pour plus d'information sur l'authentification JAAS, voir le tutoriel d'authentification JAAS.

Exemple de client

- En premier, on déclare le CallbackHandler à utiliser. Ce peut être une simple ligne d'invite de commande, un dialogue, ou même un login/password à utiliser.

Exemple de CallbackHandler qui peut être utilisé avec JOnAS.

```
CallbackHandler handler1 = new LoginCallbackHandler();
CallbackHandler handler2 = new DialogCallbackHandler();
CallbackHandler handler3 = new NoInputCallbackHandler("jonas_user", "jonas_password");
```

- Ensuite, on appelle la méthode LoginContext avec le CallbackHandler défini précédemment et l'entrée à utiliser du fichier `JONAS_ROOT/conf/jaas.config`.
  Cet exemple utilise le CallbackHandler de dialogue.

```
LoginContext loginContext = new LoginContext("jaasclient", handler2);
```

- Enfin, on appelle la méthode login sur l'instance LoginContext.

```
loginContext.login();
```

S'il n'y a pas d'exception, l'authentification est réussie.
L'authentification peut échouer si le password fourni est incorrect.

Services Web avec JOnAS

A partir de JOnAS 3.3, les Services Web peuvent être utilisés au sein des EJBs et/ou des servlets/JSPs. Cette intégration est conforme à la spécification JSR 921(v1.1) (public draft 3).
Note: Pendant que la JSR 921 est en phase de revue de maintenance, nous fournissons un lien vers la spécification 109(v1.0).

1. Web Services

A. Quelques Définitions

WSDL: WSDL (Web Service Description Language) spécifie un format XML pour décrire un service comme un ensemble de endpoints qui opèrent sur des messages.
SOAP: SOAP (Simple Object Access Protocol) v1.1, cette spécification définit un protocole basé sur XML pour l'échange d'information en environnement décentralisé et distribué. SOAP définit une convention pour représenter des appels de procédures à distance et leurs réponses.
JAXRPC: JAXRPC (Java Api for XML RPC) v1.1, est la spécification qui définit le modèle de programmation pour le RPC XML en Java.

B. Vue d'ensemble d'un Service Web

Au sens strict, un Service Web est une fonction bien définie, modulaire, encapsulée et utilisée pour une intégration à couplage lâche entre des composants applicatifs ou 'systèmes'. Il est basé sur des technologies standard, telles que XML, SOAP, et UDDI.

Les Services Web sont généralement exposés et découverts à travers un service d'annuaire standard. Avec ces standards, les clients des Services Web, qu'ils soient des utilisateurs ou d'autres applications, peuvent avoir accès à un large panel d'information −− données financières personnelles, nouvelles, la météo, et des documents d'entreprise −− au travers d'applications qui résident sur des serveurs éparpillés sur le réseau.

Les Services Web utilisent une définition WSDL (voir www.w3.org/TR/WSDL) comme un contrat entre un client et un serveur (appelé également "endpoint"). WSDL définit les types à sérialiser à travers le réseau (décrits avec XMLSchema), les messages à envoyer et à recevoir (composition, paramètres), les portTypes (vue résumée d'un Port), les "bindings" (description concrète du PortType : SOAP, GET, POST, ...), les services (ensemble des Ports), et le Port (le port est associé à une URL unique de serveur qui définit la localisation d'un Service Web).

Un Service Web au sens J2EE est un composant avec quelques méthodes exposées et accessible via HTTP (au travers de servlet(s)). Les Services Web peuvent être implémentés en temps que "Stateless Session Beans" ou en temps que classes JAXRPC (une simple classe Java, pas de nécessité d'héritage).

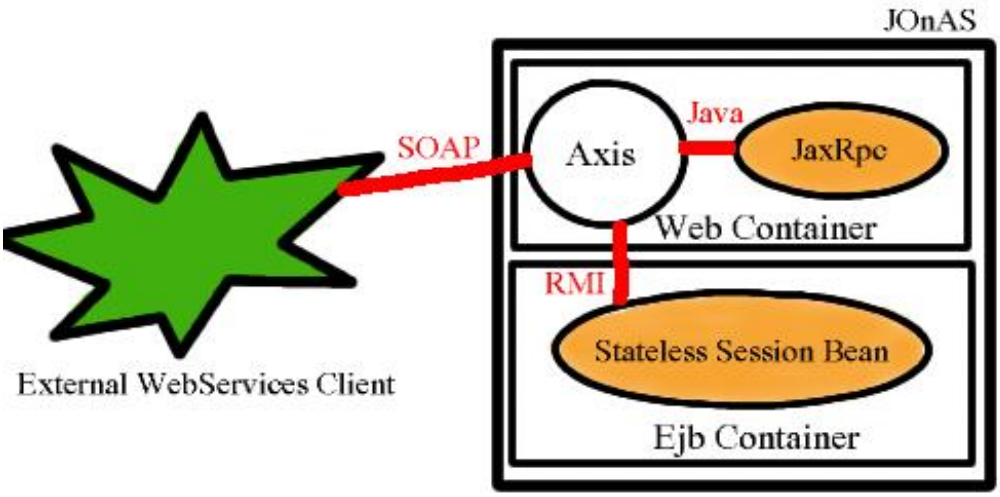


Figure 1. Les serveurs WebServices (endpoints) déployés dans JOnAS (un programme client externe peut accéder au endpoint via AxisServlet)

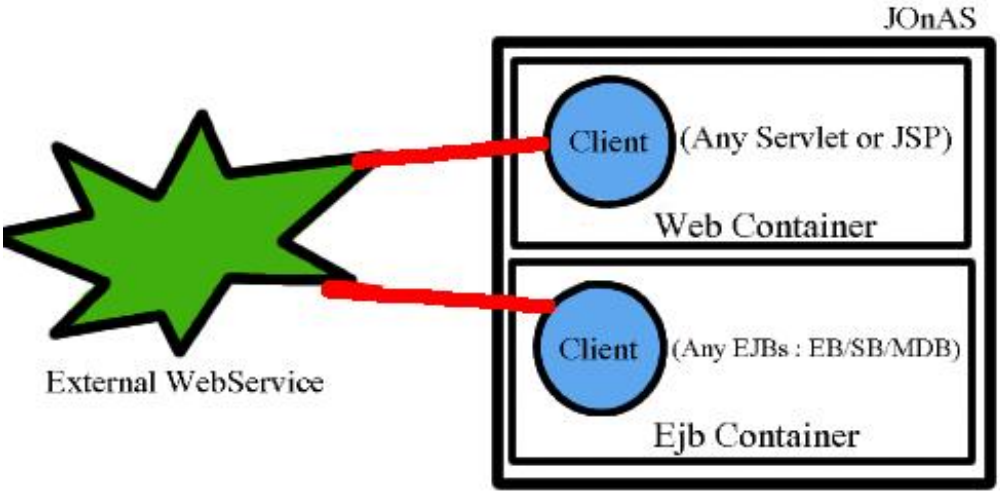


Figure 2. Client WebServices déployé dans JOnAS (peut accéder à des WebServices externes)

La servlet est utilisée pour répondre à la requête d'un client et envoyer l'appel à l'instance désignée pour y répondre (le bean SSB ou la classe JAXRPC exposée comme Service Web). Il prend en charge la dé−sérialisation du message SOAP entrant pour transformer le SOAP XML en un objet Java, exécuter l'appel, et sérialiser le résultat de l'appel (ou l'exception renvoyée) en SOAP XML avant de renvoyer le message de réponse au client.

2. Exposer un Composant J2EE en tant que Service Web

Il existe deux types de composants J2EE qui peuvent être exposés en tant que Services Web côté serveur : Les StateLess Session Beans et les classes JAX−RPC. Les serveurs (endpoints) de WebServices ne doivent pas contenir d'information d'état.

Un nouveau Descripteur de Déploiement standard a été créé pour décrire les serveurs (endpoints) de Services Web. Ce descripteur est nommé "webservices.xml" et peut être utilisé dans une "webapp" (dans WEB−INF/) ou dans un EjbJar (dans META−INF/). Ce descripteur a son Descripteur de Déploiement spécifique à JOnAS (jonas−webservices.xml est optionnel).

A cet instant, l'utilisateur doit configurer le descripteur de déploiement spécifique à Axis (.wsdd). Se référer au guide de référence d'Axis pour des informations complémentaires sur les fichiers WSDD.
Se référer à l'exemple webServices pour des fichiers exemples (répertoire etc).

A. JAX−RPC Endpoint

Un JAX−RPC endpoint est une simple classe s'exécutant dans le conteneur de servlets (Tomcat ou Jetty). Les requêtes SOAP sont envoyées à une instance de cette classe et la réponse est sérialisée et renvoyée au client.
Un JAX−RPC endpoint doit se trouver dans une WebApp (le fichier war doit contenir un "WEB−INF/webservices.xml").

Note : Le développeur doit déclarer la AxisServlet avec un seul servlet−mapping. De plus, un server−config.wsdd doit être fourni dans le répertoire WEB−INF. Ce fichier indique à Axis quel service doit être exposé (un élément service correspond à un composant port dans le fichier webservices.xml) avec les informations pour accéder à la classe du service positionnées en tant que sous−élément(s) paramètre(s) (className, allowedMethods, ...).

B. Stateless Session Bean Endpoint

Un SSB est un EJB qui va être exposé (toutes ou certaines de ses méthodes) en tant que WebService endpoint.
En règle générale, un SSB doit se trouver dans un fichier EjbJar, qui doit contenir un "META−INF/webservices.xml".

Note : Parce qu'un SSB ne peut pas être accédé directement via HTTP, l'utilisateur doit créer une webapp vide qui déclare la AxisServlet avec un fichier server−config.wsdd (voir précédemment). Pour cela, l'utilisateur doit mettre à jour l'information pour l'accès au bean (beanJndiName, localHomeInterfaceName, localInterfaceName, homeInterfaceName, remoteInterfaceName) dans server−config.xml. Ces paramètres ne sont pas tous nécessaires, seulement ceux spécifiques aux besoins de l'utilisateur (par exemple, pas d'interfaces distantes si un SSB est purement local). Le développeur doit également créer un descripteur de déploiement spécifique à JOnAS pour services web : jonas−webservices.xml. Le nom de fichier du war créé précédemment doit être précisé pour identifier pour JOnAS l' URL d'accès au SSB. Enfin, ces deux fichiers (EjbJar et WebApp) doivent être ajoutés dans une archive Application.

C. Utilisation

Dans ce descripteur, le développeur décrit les composants qui vont être exposés en tant que WebServices' endpoints ; on les appelle les composants−port (port−component). Un ensemble de composants−port définit une webservice−description, et une webservice−description utilise un fichier de Définition WSDL pour une description complète des WebServices' endpoints.

Chaque composant−port est associé au composant J2EE qui va répondre à la requête (service−impl−bean avec un sous−élément servlet−link ou ejb−link) ainsi qu'à un port WSDL (wsdl−port définissant le QName du port). Une liste des handlers JAX−RPC est fournie pour chaque composant−port. La "service−endpoint−interface", optionnelle, définit les méthodes des composants J2EE qui seront exposées (héritage pas nécessaire).

Un handler JAX−RPC est une classe utilisée pour lire et/ou modifier le message SOAP avant transmission et/ou après réception (voir la spec. JAX−RPC v1.1 chap#12 "SOAP Message Handlers"). Le Handler Session, un exemple simple, va lire/écrire un information de session dans les Headers SOAP. Les handlers sont identifiés par un nom unique (au sein de l'entité applicative), sont initialisés avec le(s) init−param(s), et s'appliquent sur une liste de Headers SOAP défini à partir des sous−éléments soap−headers. Le handler est exécuté en tant qu'acteur(s) SOAP défini dans la liste des soap−roles.

Une webservice−description définit un ensemble de composants−ports, une définition WSDL (décrivant le WebService) et un fichier de mapping (WSDL−2−Java bindings). Les éléments wsdl−file et jaxrpc−mapping−file doivent préciser un chemin vers un fichier contenu dans l'entité module (i.e., le fichier war/jar). A noter qu'une URL ne peut pas être positionnée à ce niveau. La spécification impose également que les WSDLs soit placés dans un sous−répertoire wsdl (i.e., WEB−INF/wsdl ou META−INF/wsdl) ; il n'y a pas de telle exigence en ce qui concerne le fichier jaxrpc−mapping−file. Tous les ports définis dans le WSDL doivent être associés à un composant−port. Ceci est primordial car le WSDL est un contrat entre le serveur et un client (si le client utilise un port non implémenté/associé à un composant, l 'appel client échouera systématiquement).

De même que pour tous les autres descripteurs de déploiement, une DTD standard est utilisée pour contraindre le XML. A noter que les DTDs seront remplacées par des schémas XML avec J2EE 1.4.

D. Exemple Simple (exposer un serveur JAX−RPC) de webservices.xml

```
<?xml version="1.0"?>

<!DOCTYPE webservices PUBLIC
  "−//IBM Corporation, Inc.//DTD J2EE Web services 1.0//EN"
  "http://www.ibm.com/webservices/dtd/j2ee_web_services_1_0.dtd">

<webservices>
  <display−name>Simple Web Service Endpoint DeploymentDesc</display−name>
  <webservice−description>
    <!−− name must be unique in an Application unit −−>
    <webservice−description−name>Simple Web Service Endpoint</webservice−description−

    <!−− Link to the WSDL file describing the endpoint −−>
    <wsdl−file>WEB−INF/wsdl/warendpoint.wsdl</wsdl−file>
    <!−− Link to the mapping file describing binding between WSDL and Java −−>
    <jaxrpc−mapping−file>WEB−INF/warEndpointMapping.xml</jaxrpc−mapping−file>

    <!−− The list of endpoints −−>
    <port−component>
```

```
      <!-- Unique name of the port-component -->
      <port-component-name>WebappPortComp1</port-component-name>
      <!-- The QName of the WSDL Port the J2EE port-component is linked to -->
      <wsdl-port>
        <namespaceURI>http://wsendpoint.servlets.ws.objectweb.org</namespaceURI>
        <localpart>wsendpoint1</localpart>
      </wsdl-port>
      <!-- The endpoint interface defining methods exposed for the endpoint -->

<service-endpoint-interface>org.objectweb.ws.servlets.wsendpoint.WSEndpointSei</servi
      <!-- Link to the J2EE component (servlet/EJB) implementing methods of the SEI -
     <service-impl-bean>
        <!-- name of the servlet element defining the JAX-RPC endpoint -->
        <!-- can be ejb-link if SSB is used (only in EjbJar !) -->
        <servlet-link>WSEndpoint</servlet-link>
      </service-impl-bean>

      <!-- The list of optional JAX-RPC Handlers -->
      <handler>
        <handler-name>MyHandler</handler-name>
        <handler-class>org.objectweb.ws.handlers.SimpleHandler</handler-class>
        <!-- A list of init-param for Handler configuration -->
        <init-param>
           <param-name>param1</param-name>
           <param-value>value1</param-value>
        </init-param>
        <init-param>
           <param-name>param2</param-name>
           <param-value>value2</param-value>
        </init-param>
      </handler>
    </port-component>
  </webservice-description>
</webservices>
```

E. Le fichier optionnel jonas-webservices.xml

Le fichier jonas-webservices.xml est collocalisé avec webservices.xml. C'est un Descripteur de Déploiement optionnel (nécessaire dans certains cas seulement). Son rôle est de lier un webservices.xml à la WebApp qui a en charge la diffusion de la requête SOAP. En fait, cela est nécessaire uniquement pour un EjbJar (le seul qui dépende d'une autre servlet pour être accessible avec HTTP/HTTPS) qui n'utilise pas la Convention de Nommage par Défaut utilisée pour récupérer un nom de webapp à partir du nom EjbJar.

Convention : <ejbjar-name>.jar aura une WebApp <ejbjar-name>.war .

Exemple:

```
<?xml version="1.0"?>
<!DOCTYPE jonas-webservices PUBLIC
          "-//Objectweb//DTD JOnAS WebServices 3.3//EN"
          "http://jonas.objectweb.org/dtds/jonas-webservices_3_3.dtd">
<jonas-webservices>
    <!-- the name of the webapp in the EAR (it is the same as the one in
application.xml) -->
    <war>dispatchingWebApp.war</war>
</jonas-webservices>
```

F. Modifications à jonas-web.xml

JOnAS permet aux développeurs de configurer entièrement une application en précisant le hostname, le context-root, et le port utilisé pour accéder aux WebServices. Celà est fait dans jonas-web.xml de l'application WebApp d'accès.

host: configure le hostname à utiliser dans l' URL (doit être un hôte conteneur web disponible).

port: configure le numéro de port à utiliser dans l'URL (doit être un numéro de port de connecteur HTTP/HTTPS disponible).

Quand ces valeurs ne sont pas positionnées, JOnAS va essayer de déterminer les valeurs par défaut pour l'hôte et le port.

Limitations :

- L'hôte peut être déterminé uniquement si un seul hôte est positionné comme conteneur web.
- Le port peut être déterminé uniquement si un seul connecteur est utilisé par le conteneur web.

3. Le client d'un WebService

Un EJB ou une servlet qui veut utiliser un WebService (en tant que client) doit déclarer un lien sur le WebService avec un élément service-ref (même principe que pour tous les éléments *-ref).

Note: Dans JOnAS 3.3, la grammaire XML de la spécification EJB 2.1ou servlet 2.4 n'est pas encore supportée. C'est pourquoi, pour utiliser les éléments service-ref dans un ejb-jar.xml et un web.xml, le développeur doit modifier le DOCTYPE des Descripteurs de Déploiement.

Pour web.xml :

```
    <!DOCTYPE web-app PUBLIC
          "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
          "http://jonas.objectweb.org/dtds/web-app_2_3_ws.dtd">
```

Pour ejb-jar.xml :

```
    <!DOCTYPE ejb-jar PUBLIC
          "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
          "http://jonas.objectweb.org/dtds/ejb-jar_2_0_ws.dtd">
```

A. L'élément service−ref

L'élément service−ref déclare une référence à un WebService utilisé par un composant J2EE dans le Descripteur de Déploiement web et EJB.

Le composant utilise un nom logique appelé un service−ref−name pour consulter l'instance du service. Ainsi, tout composant réalisant une consultation sur un WebService doit déclarer un lien (l'élément service−ref) dans le descripteur de déploiement standard (web.xml ou ejb−jar.xml).

Exemple de service−ref:

```
  <service-ref>
    <!-- (Optional) A Web services description that can be used in
administration tool.  -->
    <description>Sample WebService Client</description>

    <!-- (Optional) The WebService reference name. (used in Administration
tools) -->
    <display-name>WebService Client 1</display-name>

    <!-- (Optional) An icon for this WebService. -->
    <icon> <!-- ... --> </icon>

    <!-- The logical name for the reference that is used in the client source
code.
         It is recommended, but not required that the name begin with
'services/' -->
    <service-ref-name>services/myService</service-ref-name>

    <!-- Defines the class name of the JAX-RPC Service interface that the
client depends on.
         In most cases, the value will be :        javax.xml.rpc.Service
         but a generated specific Service Interface class may be specified
         (require WSDL knowledge and so the wsdl-file element). -->
    <service-interface>javax.xml.rpc.Service</service-interface>

    <!-- (Optional) Contains the location (relative to the root of
         the module) of the web service WSDL description.
         - need to be in the wsdl directory.
         - required if generated interface and sei are declared. -->
    <wsdl-file>WEB-INF/wsdl/stockQuote.wsdl</wsdl-file>

    <!-- (Optional) A file specifying the correlation of the WSDL definition
         to the interfaces (Service Endpoint Interface, Service Interface).
         - required if generated interface and sei (Service Endpoint Interface)
are declared.-->
```

```
    <jaxrpc-mapping-file>WEB-INF/myMapping.xml</jaxrpc-mapping-file>

    <!-- (Optional) Declares the specific WSDL service element that is being
referred to.
         It is not specified if no wsdl-file is declared or if WSDL contains
only 1 service element.
         A service-qname is composed by a namespaceURI and a localpart.
         It must be defined if more than 1 service is declared in the WSDL. -->
    <service-qname>
      <namespaceURI>http://beans.ws.objectweb.org</namespaceURI>
      <localpart>MyWSDLService</localpart>
    </service-qname>

    <!-- Declares a client dependency on the container to resolving a Service
Endpoint Interface
         to a WSDL port. It optionally associates the Service Endpoint
Interface with a
         particular port-component. -->
    <port-component-ref>

<service-endpoint-interface>org.objectweb.ws.beans.ssbendpoint.MyService</service-enc
      <!-- Define a link to a port component declared in another unit of the
application -->

<port-component-link>ejb_module.jar#PortComponentName</port-component-link>
    </port-component-ref>

    <!--A list of Handler to use for this service-ref  -->
    <handler>
       <!-- Must be unique within the module. -->
       <handler-name>MyHandler</handler-name>

       <handler-class>org.objectweb.ws.handlers.myHandler</handler-class>

       <!-- A list of init-param (couple name/value) for Handler initialization
-->
       <init-param>
         <param-name>param_1</param-name>
         <param-value>value_1</param-value>
       </init-param>

       <!-- A list of QName specifying the SOAP Headers the handler will work
on.
            - namespace and locapart values must be found inside the WSDL. -->
       <soap-header>
```

```
         <namespaceURI>http://ws.objectweb.org</namespaceURI>
         <localpart>MyWSDLHeader</localpart>
       </soap-header>

       <!-- A list of SOAP actor definition that the Handler will play as a
role.
            a soap-role is a namespace URI. -->
       <soap-role>http://actor.soap.objectweb.org</soap-role>

       <!-- A list of port-name element defines the WSDL port-name that a
handler
            should be associated with.
            If no port-name is specified, the handler is assumed to be
associated with all ports of the service-ref. -->
       <port-name>myWSDLPort</port-name>
    </handler>
  </service-ref>
```

B. L'élément jonas-service-ref

Un jonas-service-ref doit être précisé pour chaque service-ref déclaré dans le Descripteur de Déploiement standard. Le jonas-service-ref rajoute des informations spécifiques à JOnAS (et spécifiques au moteur de WebService) relatives à l'élement service-ref.

Exemple de jonas-service-ref:

```
    <jonas-service-ref>

      <!-- Define the service-ref contained in the component
           deployment descriptor (web.xml or ejb-jar.xml).
           used as a key to associate a service-ref to its correspondent
jonas-service-ref-->
      <service-ref-name>services/myService</service-ref-name>

      <!-- Define the physical name of the resource. -->
      <jndi-name>webservice_1</jndi-name>

      <!-- A list of init-param used for specific configuration of the service
-->
      <jonas-init-param>
        <param-name>serviceClassName</param-name>
        <param-value>org.objectweb.ws.services.myServiceImpl</param-value>
      </jonas-init-param>
```

```
    </jonas-service-ref>
```

6. Limitations

- Le fichier jaxrpx-mapping-file n'est utilisé que pour récupérer le namespace XML du mapping de l'information du package java. Aucune autre information n'est utilisée à ce niveau (limitation Axis).
- service-endpoint-interface dans port-component et port-component-ref est lu, mais pas utilisé.
- l'élément port-component-link n'est pas supporté.

last update : 17 September 2003

Howto: JOnAS Versions Migration Guide

The content of this guide is the following:

JOnAS 3.1 to JOnAS 3.1.4

Applications developed for JOnAS 3.1 do not require changes; however, they should be redeployed (GenIC). The migration affects only certain customized configuration files and build.xml files.

The main changes are in the area of ***communication protocols*** support, due to the integration of CAROL. This implies the following configuration changes:

- The jndi.properties file is replaced by a carol.properties file (in JONAS_BASE/conf or JONAS_ROOT/conf) and is no longer searched for within the classpath.
- The OBJECTWEB_ORB environment variable no longer exists.
- Security context propagation is specified in the jonas.properties file, which replaces the `-secpropag` option of GenIC or the `secpropag` attribute of the JOnAS ejb-jar ANT task.
- EJBs can be deployed for several protocols, which is specified by the new option `-protocols` of GenIC or new attribute `protocols` of the JOnAS ejb-jar ANT task; previously, the protocol was chosen through the

OBJECTWEB_ORB environment variable.
- The ${OBJECTWEB_ORB}_jonas.jar files, i.e. RMI_jonas.jar or JEREMIE_jonas.jar, no longer exist; there is only one jonas.jar file.
- The previous items involve changes in application build.xml files.

Refer to the JOnAS Configuration Guide for details about Communication Protocols configuration.

Other configuration changes are due to ***security*** enhancements:

- The files tomcat−users.xml, jonas−users.properties, and jettyRealm.properties, are suppressed and replaced by a jonas−realm.xml file. This file contains the list of users/password/roles for the Memory realm, as well as the access configuration for Database and LDAP realms. Realms declared in this file have corresponding resources bound in the registry, and MBeans to be managed.
- The security service should be launched after the dbm service (order in the jonas.services property).
- A new realm with a reference to the JOnAS resource specified in the jonas−realm.xml file is used in the server.xml file (Tomcat) or in the web−jetty.xml file (Jetty).
- The jonas.properties file contains a new line specifying the jndi name of a resource (ejbrealm) that provides Java access to the user identification repository (memory, ldap, or database) of the corresponding realm (specified in the jonas−realm.xml file). This is primarily used by Java clients that intend to build their SecurityContext.

Refer to the JOnAS Configuration Guide for details about Security configuration.

The ***preferred steps*** for migrating from JOnAS 3.1 are the following:

1. Create a new JOnAS_BASE (e.g. through the ANT `create_jonasbase` target).
2. Copy the new as well as any customized files from the old JONAS_BASE to the new one, conforming to the new configuration rules (jndi.properties replaced by carol.properties, security context propagation and realm specified in jonas.properties, new realm specification in server.xml, changes in your build.xml files, content of tomcat−users.xml, jonas−users.properties or jettyRealm.properties should migrate into jonas−realm.xml).

Details for migrating a configuration are provided in the following sections.

carol.properties

Modify this file according to the content of the old jndi.properties file. If the OBJECTWEB_ORB was RMI, set `carol.protocols` to `jrmp`; if the OBJECTWEB_ORB was JEREMIE, set `carol.protocols` to `jeremie`. Then, configure the URL with host name and port number. Example:

```
carol.protocols=jrmp
carol.jrmp.url=rmi://localhost:1099
```

jonas.properties

If EJB security was used, the security context propagation should be activated. A realm resource can be chosen to be accessed from Java; this is now specified in the jonas.properties file:

```
jonas.security.propagation              true
jonas.service.security.ejbrealm         memrlm_1
jonas.services  registry,jmx,jtm,dbm,security,jms,ejb,web,ear
```

server.xml

Choose the memory, database, or ldap realm resource for Tomcat authentication.

```
<Realm className="org.objectweb.jonas.security.realm.JRealmCatalina41" debug="99" resourceName="
```

web−jetty.xml

This file is located in the WEB−INF directory of a WAR file and contains a reference to the JOnAS Realm to be used for authentication.

```
<Call name="setRealmName">
  <Arg>Example Basic Authentication Area</Arg>
</Call>
<Call name="setRealm">
  <Arg>
    <New class="org.objectweb.jonas.security.realm.JRealmJetty42">
      <Arg>Example Basic Authentication Area</Arg>
      <Arg>memrlm_1</Arg>
    </New>
  </Arg>
</Call>
```

Deployment

For existing scripts that call GenIC for deploying EJBs, the `−secpropag` option no longer exists (security propagation is activated from the jonas.properties file as illustrated previously), and a new option `−protocols` specifies a comma−separated list of protocols (chosen within jeremie, jrmp, iiop, cmi) for which stubs will be generated. The default value is `jrmp,jeremie`.

```
GenIC −protocols jrmp,jeremie,iiop
```

Refer to the following for the deployment ANT task.

build.xml files

The build.xml files for building JOnAS examples have been upgraded according to the new configuration scheme. Existing build.xml files must be updated the same way:

- `<property name="orb" value="${myenv.OBJECTWEB_ORB}" />` is no longer used and must be suppressed.
- In the target building the classpath, replace `${orb}_jonas.jar` by `jonas.jar.`
- In the jonas deployment task, suppress the attributes `orb="${orb}" secpropag="yes,"` and add the attribute `protocols="${protocols.names}."` The build.properties file of the JOnAS examples now contains `protocols.names=jrmp,jeremie`.

JOnAS 3.0 to JOnAS 3.1

Applications developed for JOnAS 3.0 can be redeployed without any changes.

The differences in the execution environment are the following:

- JOnAS is available under three different packagings.
- The location and the JOnAS configuring policy has changed.
- The location and the policy to deploy applications has changed.

JOnAS is still available as a "single ejb container" as before. Additionally, two new packagings as "J2EE server" are available:

- jonas3.1−tomcat4.1.24 package
- jonas3.1−jetty4.2.9 package.

For these two new packagings, it is no longer necessary to set the environment variable CATALINA_HOME or JETTY_HOME. These packagings have JOnAS examples compiled for use with RMI.

The location and the policy for JOnAS configuration has changed:

- Configuration files are located under $JONAS_ROOT/conf (in previous versions they were located under $JONAS_ROOT/config).
- The old policy used to search for configuration files (working directory, home directory, $JONAS_ROOT/config) is no longer used. The new policy is the following:
  If the environment variable JONAS_BASE is set, configuration files are searched for in $JONAS_BASE/conf, if not, under $JONAS_ROOT/con.

The location and the policy for deploying applications has changed:

- If the environment variable JONAS_BASE is set, the application to be deployed with `jadmin` or `jonas admin` are searched for in $JONAS_BASE/(ejbjars|apps|webapps), if not, under

$JONAS_ROOT/(ejbjars|apps|webapps).

JOnAS 2.6.4 to JOnAS 3.0

Application with entity beans CMP 1.1

The standard deployment descriptor must be updated for applications deployed in previous versions of JOnAS using entity beans that have container−managed persistence CMP 1.1. For such entity beans, a `<cmp−version> tag` with value `1.x` must be added. For example a deployment descriptor that looks like the following:

```
<persistence−type>container</persistence−type>
<cmp−field>
  <field−name>fieldOne</field−name>
</cmp−field>
<cmp−field>
  <field−name>fieldTwo</field−name>
</cmp−field>
```

must be changed as follows:

```
<persistence−type>container</persistence−type>
<cmp−version>1.x</cmp−version>
<cmp−field>
  <field−name>fieldOne</field−name>
</cmp−field>
<cmp−field>
  <field−name>fieldTwo</field−name>
</cmp−field>
```

The EJB 2.0 specification states that the default value of the `<cmp−version> tag` is 2.x.

JOnAS 2.6 to JOnAS 2.6.1

Installation procedure

Before using JOnAS when it is installed on a host, the `ant install` must be run in the JOnAS_ROOT directory to rebuild global libraries based on your environment (RMI/JEREMIE, web environment). This must be run again if the orb (i.e. from RMI to JEREMIE) is switched, or if the web environment (i.e. switching from CATALINA to JETTY) changes.

EJB container creation

It is still possible to create an EJB container from an EJB deployment descriptor identified by its *xml* file name in JOnAS version 2.6.1 (as required by JOnAS users with versions prior to JOnAS 2.6).

JOnAS 2.5 to JOnAS 2.6

Use of Make discontinued for building

The *make* tool is no longer used to build JOnAS and the JOnAS examples. *Common makefiles* previously used to build the JOnAS examples via *make* are no longer delivered in `$JONAS_ROOT/gmk`.
Ant is the only build tool used in JOnAS 2.6.

Building with Ant 1.5

In JOnAS 2.6, the required version of Ant is 1.5 (instead of Ant 1.4).
In addition, starting with this version, an ant task *ejbjar* for JOnAS is delivered in the JOnAS distribution. Refer to the JOnAS example to see how this new task is used.

New jonas command

A new **`jonas`** command is delivered that provides the capability to:

- start a JOnASserver,
- stop a JOnASserver,
- administrate a JOnASserver,
- check the JOnASenvironment,
- print the JOnASversion.

The `EJBServer`, `JonasAdmin` and `CheckEnv` commands are now deprecated.
Refer to the JOnAS Commands Reference Guide in the JOnAS documentation for details.

EJB container creation

In previous versions of JOnAS, an EJB container could be created from an ejb−jar file or from an EJB deployment descriptor identified by its *xml* file name.
In JOnAS version 2.6, an EJB container can only be created from an ejb−jar file.
This means that an *xml* file name can no longer be specified as:

- a value of the `jonas.service.ejb.descriptors` jonas property,
- or an argument of the `−a` option of the `jonas admin` command.

Tomcat 4.0.x Support

Tomcat version 4.0.1 is no longer supported. The current supported versions are Tomcat 4.0.3 and Tomcat 4.0.4.

Tomcat Service and Web Container Service

JOnAS 2.6 is a full−featured J2EE application server, thus Tomcat can be used with JOnAS as the Web container. This functionality is set up via the JOnAS service *web*. In earlier version of JOnAS, a service *tomcat* was provided. This service *tomcat* is now deprecated because it was not compliant with J2EE specification. Note that if the *tomcat* service was used previously and the *web* container service is now being used, the war files (and expanded directories) that were deployed here (they are now deployed in JONAS_ROOT/webapps) should be suppressed from the CATALINA_HOME/webapps directory.

Refer to the section "Integrating Tomcat and JOnAS" for details.

JOnAS 2.4.4 to JOnAS 2.5

Tomcat 4.0.x Support

JOnAS 2.5 supports two versions of Tomcat for the tomcat service: Tomcat 3.3.x (as in JOnAS 2.4.4) and Tomcat 4.0.x. The default support of JOnAS 2.4.4 is for Tomcat 3.3.x, therefore nothing needs to be changed if the tomcat service was being used with JOnAS 2.4.4. For Tomcat 4.0.x, the `jonas.service.tomcat.class` property of `jonas.properties` file to `org.objectweb.jonas.tomcat.EmbeddedTomcatImpl40` must be set. Refer to the How to use Tomcat with JOnAS document for more details.

trace.properties

Starting with JOnAS 2.5, the Objectweb Monolog logging API is used for JOnAS traces. The JOnAS traces configuration file `trace.properties` has changed. Refer to the Configuration Guide in the JOnAS documentation for details.

JOnAS 2.4.3 to JOnAS 2.4.4

The main differences between these two versions are the two new services provided in JOnAS 2.4.4: **registry** and **tomcat**.

	JOnAS 2.4.3	**from JOnAS 2.4.4**
jonas services	• jonas.services *jmx,security,jtm,dbm,resource,jms,ejb*	• jonas.services *registry,tomcat,jmx,security,jtm,dbm,resource,jms,ejb*
Registry service configuration	–	• jonas.service.registry.class *org.objectweb.jonas.registry.RegistryServiceImpl* • jonas.service.registry.mode *automatic*
Tomcat service configuration	–	• jonas.service.tomcat.class *org.objectweb.jonas.tomcat.EmbeddedTomcatImpl33*

		• jonas.service.tomcat.args

JOnAS 2.3 to JOnAS 2.4

Introduction

This chapter is intended for JOnAS users migrating applications from JOnAS 2.3 to JOnAS 2.4 and later versions. This migration does not affect EJB components' files. However, two configuration files are slightly different: the jonas.properties file and the jonathan.xml file.

- **jonas.properties:** due to the new JOnAS architecture regarding services (refer to the advanced topic chapter on JOnAS services in the JOnAS documentation), the structure of the properties defined in this file has changed. It is necessary to upgrade a jonas.properties file written for a version 2.x (x<4) to reuse it for JOnAS 2.4.
- **jonathan.xml:** for applications using the JEREMIE distribution mechanism, it is necessary to upgrade this configuration file, since JOnAS has embedded a new version of Jonathan.

Upgrading your jonas.properties file

JOnAS EJB servers are configured via the jonas.properties file. This configuration file may be located in three different places:

1. $JONAS_ROOT/config/jonas.properties
2. $HOME/jonas.properties: the home directory
3. ./jonas.properties: the directory from which the EJB server is launched.

An EJB server reads the three potential files in this order listed (1, 2, 3), each one possibly overwriting properties defined in a previous file. Therefore, existing jonas.properties files from previous JOnAS versions must be upgraded in order to retain the configuration settings, by making the following structural changes:

	before JOnAS 2.4	**from JOnAS 2.4**
jonas services (new)	_	• jonas.services *jmx,security,jtm,dbm,resource,jms,ejb*
JMX service configuration	_	• jonas.service.jmx.class *org.objectweb.jonas.jmx.JmxServiceImpl*
JOnAS EJB service configuration (beans to be loaded)	• jonas.beans.descriptors ...	• jonas.service.ejb.class *org.objectweb.jonas.container.EJBServiceImpl* • jonas.service.ejb.descriptors ...
JOnAS DBM service	• jonas.datasources ...	

configuration		• jonas.service.dbm.class *org.objectweb.jonas.dbm.DataBaseServiceImpl* • jonas.service.dbm.datasources ...
JOnAS JTM service configuration	• jonas.tm.remote *false* • jonas.tm.timeout *60*	• jonas.service.jtm.class *org.objectweb.jonas.jtm.TransactionServiceImpl* • jonas.service.jtm.remote *false* • jonas.service.jtm.timeout *60*
JOnAS SECURITY service configuration	–	• jonas.service.security.class *org.objectweb.jonas.security.JonasSecurityServiceImpl*
JOnAS JMS service configuration	• jonas.jms.mom *org.objectweb.jonas_jms.JmsAdminForJoram* • jonas.jms.collocated *true* • jonas.jms.url *joram://localhost:16010* • jonas.jms.threadpoolsize *10* • jonas.jms.topics *sampleTopic* • jonas.jms.queues ...	• jonas.service.jms.class *org.objectweb.jonas.jms.JmsServiceImpl* • jonas.service.jms.mom *org.objectweb.jonas_jms.JmsAdminForJoram* • jonas.service.jms.collocated *true* • jonas.service.jms.url *joram://localhost:16010* • jonas.service.ejb.mdbthreadpoolsize *10* • jonas.service.jms.topics *sampleTopic* • jonas.service.jms.queues ...
JOnAS RESOURCE service configuration (Resource Adapters to be installed)	–	• jonas.service.resource.class *org.objectweb.jonas.resource.ResourceServiceImpl* • jonas.service.resource.resources ...

The main transformation rule is that most of the properties are now part of a JOnAS service. For each service XXX, the class property *jonas.service.XXX.class* containing the name of the service class (all these class properties are set in the $JONAS_ROOT/config/jonas.properties file) must be specified, and each additional property *p* related to the service is named *jonas.service.XXX.p*. The list of services to be launched with the server is specified in the *jonas.services* property. These services are EJB (in which are defined the beans to be loaded), JTM (in which are defined the transaction monitor properties), DBM (in which are defined the datasources), SECURITY, JMS (the messaging service), and JMX (a new service for management).

Upgrading your Jonathan configuration file

In the new version of Jonathan, the jonathan.prop has been replaced by jonathan.xml.

Howto: Installing JOnAS from scratch

This guide provides instructions for installing JOnAS from scratch on Unix−compatible systems.

The content is organized into the following steps

1. JDK 1.3 or 1.4 installation
2. Ant 1.5 installation
3. Tomcat 4.1.x installation
4. JOnAS installation
5. Setup

JDK 1.4 or 1.3 installation

Download the binary version of JDK 1.3 or JDK 1.4 from the java Sun web site into the appropriate directory. Launch the executable file:

`./j2sdk−1_<version number>−<system>.bin` for Unix
`./j2sdk−1_<version number>−<system>.sh` for Linux
`j2sdk−1_<version number>−windows−i586.exe` for Windows

Set the JAVA_HOME environment variable and update the path:

```
export JAVA_HOME=<Installation Directory>
PATH=$JAVA_HOME/bin:$PATH (on Windows: PATH=%JAVA_HOME%/bin;%PATH%)
```

Ant 1.5 installation

Download the binary version of Ant 1.5 from the Ant Apache web site. Untar or Unzip it into the appropriate directory:

```
tar −jxvf jakarta−ant−1.5−bin.tar.bz2
```
(*or* *`unzip jakarta−ant−1.5−bin.zip`*)

Set the ANT_HOME environment variable and update the path:

```
export ANT_HOME=<Installation Directory>
PATH=$PATH;$ANT_HOME/bin (on Windows: PATH=%ANT_HOME%/bin;%PATH%)
```

Download bcel−5.0.zip from the Jakarta web site and install bcel.jar in the directory `$ANT_HOME/lib.`

Tomcat 4.1.x installation

Download the binary version of Tomcat 4.1.x from the Jakarta Tomcat web site. Untar it into the appropriate directory:

```
tar −xvf jakarta−tomcat−4.1.x.tar.gz
```

Set the CATALINA_HOME environment variable:

```
export CATALINA_HOME=<Installation Directory>
```

JONAS_BASE directory can be used as CATALINA_BASE: `export CATALINA_BASE=$JONAS_BASE`
Configuration information for the Realm and users is provided at the Setup process.

JOnAS installation

Download the binary version of JOnAS from the ObjectWeb web site.
Choose a location for the JOnAS installation.
Be aware that if you have already installed a previous version of JOnAS in this location, the new installation will overwrite existing files, thus customized configuration files may be lost. Therefore, it is prudent to save these files before starting the installation process.
The installation process consists of untaring the downloaded file.
Change to the directory in which JOnAS is to be installed and untar this file, using the `tar −zxvf jonas.tgz` command.
After installing the JOnAS product, set the following environment variable:

```
export JONAS_ROOT = <Installation Directory>
PATH = $JONAS_ROOT/bin/unix:$PATH
```

Do an `ant install` in the `JONAS_ROOT` directory to unpack the jar files necessary to build `jonas.jar` and `client.jar`.

Setup

Before using JOnAS, the following setup activities must be completed:

- Based on the data source being used, create a file `<data source>.properties` (templates are located in the directory `$JONAS_ROOT/conf`). Then, add the data source file name (without the extension `.properties`) to the `jonas.properties` file:
  `jonas.service.dbm.datasource <data source>`
  In the file `$JONAS_ROOT/bin/unix/config_env`, set the path to the JDBC driver of the database being used.
  For example, for PostgreSQL:
  `CLASSPATH=/usr/lib/pgsql/jdbc7.0−1.2.jar:$CLASSPATH`

- JONAS_BASE directory can be used as CATALINA_BASE: `export CATALINA_BASE=$JONAS_BASE` so it will use the JOnAS realms.
- If required, configure the Mail service (for PetStore or the example mailsb for instance). Two types of files that can be adapted to fit your installation are located in the directory `$JONAS_ROOT/conf`: `MailSession1.properties` and `MailMimePartDS1.properties`. Then, in the `jonas.properties` file, define the `jonas.service.mail.factories` property:
  ```
  jonas.service.mail.factories     MailSession1,MailMimePartDS1
  ```

Compile the examples as follows:
`ant install` in the directory `$JONAS_ROOT/examples`

JOnAS installation is now complete. For more information, refer to the JOnAS Documentation.

Howto: Installing the packaging JOnAS with a web container (JOnAS/Tomcat or JOnAS/Jetty) from scratch

This guide provides instructions for installing JOnAS (with a web container already included) from scratch.

The content is organized into the following steps:

JDK 1.3 or 1.4 installation

Download the binary version of JDK 1.3 or JDK 1.4 from the java Sun web site into the appropriate directory. Launch the executable file:

`./j2sdk-1_<version number>-<system>.bin` for Unix
`./j2sdk-1_<version number>-<system>.sh` for Linux
`j2sdk-1_<version number>-windows-i586.exe` for Windows

Set the JAVA_HOME environment variable and update the path:

```
export JAVA_HOME=<Installation Directory>
PATH=$JAVA_HOME/bin:$PATH (on Windows : PATH=%JAVA_HOME%/bin;%PATH%)
```

ANT 1.5 installation

Download the binary version of Ant 1.5 from the Ant Apache web site. Untar or Unzip it into the appropriate directory:

```
tar -jxvf jakarta-ant-1.5-bin.tar.bz2
```
(*or* *`unzip jakarta-ant-1.5-bin.zip`*)

Set the ANT_HOME environment variable and update the path:

```
export ANT_HOME=<Installation Directory>
PATH=$PATH;$ANT_HOME/bin (on Windows : PATH=%ANT_HOME%/bin;%PATH%)
```

Download bcel-5.0.zip from the Jakarta web site and install bcel.jar in the directory `$ANT_HOME/lib.`

JOnAS/Web Container installation

Download the binary version of JOnAS with Tomcat or Jetty from the ObjectWeb forge web site.
Choose a location for the JOnAS installation.
Be aware that if you have already installed a previous version of JOnAS in this location, the new installation will overwrite the existing files, thus customized configuration files may be lost. Therefore, it is prudent to save these files before starting the installation process.
The installation process consists of untaring the downloaded file.
Change to the directory in which JOnAS will be installed and untar this file, using
the `tar -zxvf jonas.tgz` command. Note that this file can be opened with winzip on Windows.
After installing the JOnAS product, set the following environment variable:

```
export JONAS_ROOT = <Installation Directory>
PATH = $JONAS_ROOT/bin/unix:$PATH (on Windows: PATH=%JONAS_ROOT%/bin/nt;%PATH%)
```

Setup

Before using JOnAS, complete the following setup activities:

- If a `CATALINA_HOME` or `CATALINA_BASE` or `JETTY_HOME` environment variable has already been set, it should be unset. JOnAS will set these variables, without requiring any modifications.
- Based on the data source being used, create a file `<data source>.properties` (templates are located in the directory `$JONAS_ROOT/conf`). Then add the data source file name (without the extension `.properties`) to the `jonas.properties` file:
  ```
  jonas.service.dbm.datasource      <data source>
  ```
  In the file `$JONAS_ROOT/bin/unix/config_env`, set the path to the JDBC driver of the database being used.
  For example, for PostgreSQL :
  ```
  CLASSPATH=/usr/lib/pgsql/jdbc7.0-1.2.jar:$CLASSPATH
  ```
- If required, configure the Mail service (for PetStore or the example mailsb, for example). JOnAS provides two types of mail factories: `javax.mail.Session` and `javax.mail.internet.MimePartDataSource`. Two types of files that can be adapted to fit your installation are located in the directory `$JONAS_ROOT/conf`: `MailSession1.properties` and `MailMimePartDS1.properties`. Then, in the `jonas.properties` file, define the `jonas.service.mail.factories` property:
  ```
  jonas.service.mail.factories      MailSession1,MailMimePartDS1
  ```

Starting JOnAS and running some examples

If the **Setup step** has not been completed described, JOnAS may not **work**.
Use the command **`jonas check`** to verify that the environment is correct.

If the environment is **correct**, JOnAS is **ready** to use.

Do a **`jonas start`**, then use a browser to go http://localhost:8080/. (*Modify this url with the appropriate hostname*.)
From the root context in which JOnAS was deployed, you can execute the earsample, access the JOnAS administration application, as well as perform other functions.
For more information, consult the JOnAS Documentation.
Follow the getting started guide to run the examples provided with JOnAS.

Howto: How to compile JOnAS

The content of this guide is the following:

Target Audience and Rationale

The target audience for this chapter is the JOnAS user who wants to build a JOnAS version from the source code obtained from CVS.

Getting the JOnAS Source

CVS (Concurrent Version System) provides network−transparent source control for groups of developers. It runs on most UNIX systems and on Windows NT systems. Refer to http://www.cyclic.com for more information

CVS provides many *read−only cvs* commands, such as `cvs status` or `cvs diff`. However, because it is read only, an individual user cannot commit changes. To start working with CVS on JOnAS, make a `checkout` of the `jonas` module, using the following command:

```
cvs −d :pserver:anonymous@cvs.jonas.forge.objectweb.org:/JOnAS login
(hit enter key when prompted for password)
cvs −d :pserver:anonymous@cvs.jonas.forge.objectweb.org:/JOnAS co jonas
```

The `CVSROOT` variable can be set, instead of using the ***−d*** option.

Recompiling JOnAS from the Source

1. Download *Ant* from the Ant project site.

   The `build.xml` file used for building JOnAS is located in the `objectweb/jonas` directory.
2. The JDK and ANT must have been successfully configured. (`JAVA_HOME, ANT_HOME, PATH` environment variables)
3. Set the following environment variables with the appropriate value:
   - JONAS_ROOT: mandatory
   - CATALINA_HOME: optional (mandatory for using the web container Tomcat)
   - JETTY_HOME: optional (mandatory for using the web container Jetty)
4. Perform the following:

cd $OBJECTWEB_HOME/jonas and choose a target:

- **ant install** to install a JOnAS binary distribution version in the JONAS_ROOT directory.
- **ant all** to build JOnAS and keep the files in the output directory.
- **ant archive** to build a JOnAS archive (.tgz extension) that contains a binary version of JOnAS. The archive is built in the HOME directory.

Recompiling the package JOnAS/Jetty/Axis from the Source

1. Download *Ant* from the Ant project site.

   The build.xml file used for building JOnAS is located in the objectweb/jonas directory.
2. Place the bcel.jar in the ANT_HOME/lib directory. It is available for downloading from the Jakarta web site.
3. The JDK and ANT must have been successfully configured. (JAVA_HOME, ANT_HOME, PATH environment variables)
4. Set the following environment variables with the appropriate value:
   - JONAS_ROOT: mandatory
   - JETTY_HOME: mandatory
5. Perform the following:

   cd $OBJECTWEB_HOME/jonas and choose a target:

   - **ant install_jetty** to install a JOnAS/Jetty/Axis binary distribution version into your JONAS_ROOT directory.
   - **ant all_jetty** to build JOnAS/Jetty/Axis and keep the files in the output directory.
   - **ant archive_jetty** to build a JOnAS/Jetty/Axis archive (.tgz extension) that contains a binary version of JOnAS/Jetty/Axis. The archive is built into the HOME directory.

Recompiling the package JOnAS/Tomcat/Axis from the Source

1. Download *Ant* from the Ant project site.

   The build.xml file used for building JOnAS is located in the objectweb/jonas directory.
2. Place the bcel.jar in the ANT_HOME/lib directory. It is available for downloading from the Jakarta web site.
3. The JDK and ANT must have been successfully configured. (JAVA_HOME, ANT_HOME, PATH environment variables)
4. Set the following environment variables with the appropriate value:
   - JONAS_ROOT: mandatory
   - CATALINA_HOME: mandatory
5. Perform the following:

   cd $OBJECTWEB_HOME/jonas and choose a target:

- **`ant install_tomcat`** to install a JOnAS/Tomcat/Axis binary distribution version into the `JONAS_ROOT` directory.
- **`ant all_tomcat`** to build JOnAS/Tomcat/Axis and keep the files in the `output` directory.
- **`ant archive_tomcat`** to build a JOnAs/Tomcat/Axis archive (.tgz extension) that contains a binary version of JOnAS/Tomcat/Axis. The archive is built in the `HOME` directory.

Howto: Clustering with JOnAS

This guide describes how to configure Apache, Tomcat, and JOnAS to install a cluster.

This configuration uses the Apache/Tomcat plug−in mod_jk. This plug−in allows use of the Apache HTTP server in front of one or several Tomcat JSP/Servlet engines, and provides the capability of forwarding some of the HTTP requests (typically those concerning the dynamic pages, i.e. JSP and Servlet requests) to Tomcat instances.

It also uses the In−Memory−Session−Replication technique based on the group communication protocol JavaGroups to provide failover at servlet/JSP level.

For the load balancing at EJB level, a clustered JNDI called cmi is used.

This document describes one architecture with all the clustering functionalities available in JOnAS, the configuration of architectures integrating one of those functionalities, and other possible configurations.

The content of this guide is the following:

Architecture

The architecture with all the clustering functionality available in JOnAS is: Apache as the front−end HTTP server, JOnAS/Tomcat as J2EE Container, and a shared database.

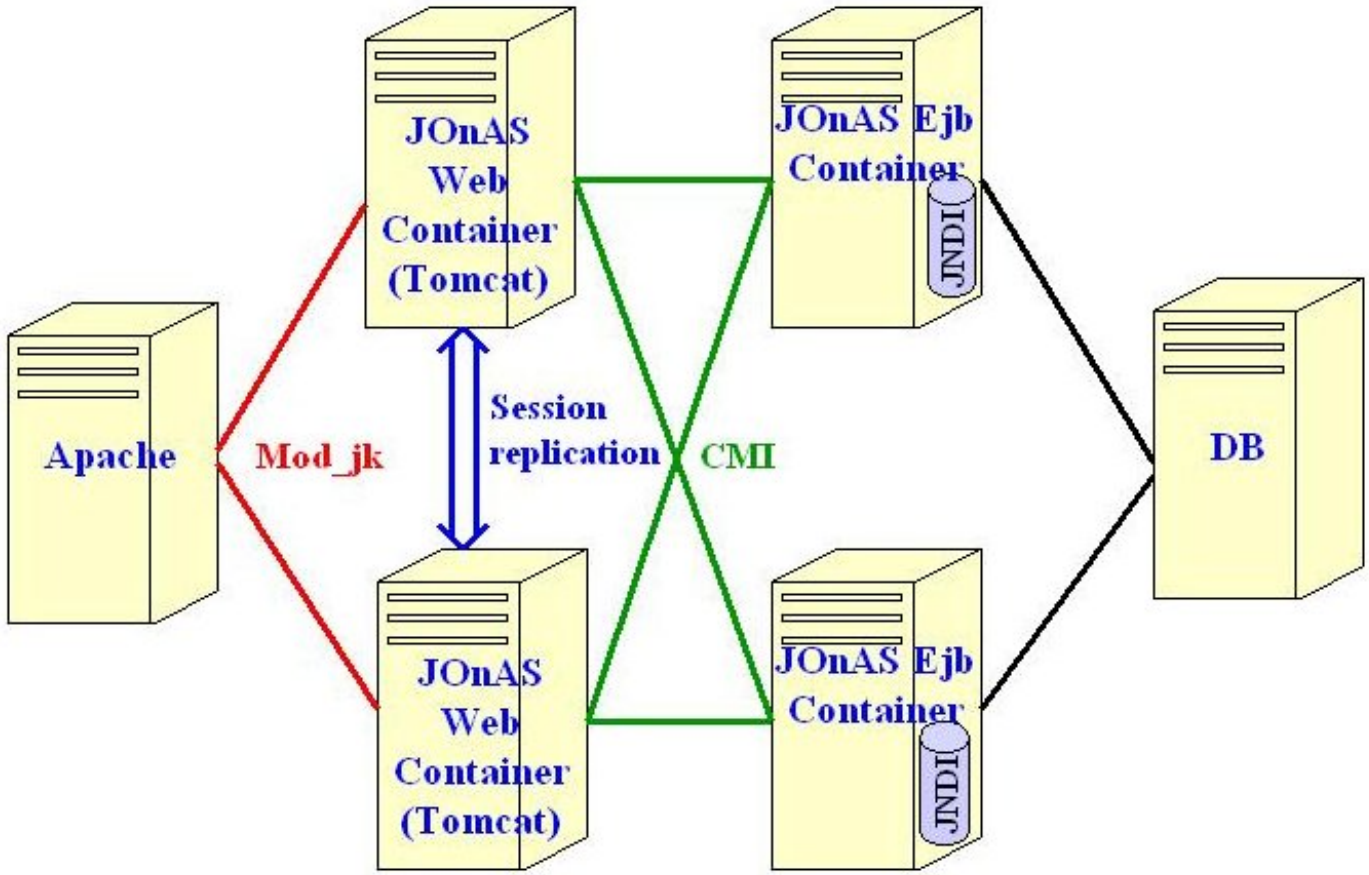
At the Servlet / JSP level, the mod_jk plug−in provides Load Balancing / High Availability and the Tomcat−Replication module provides Failover.

At the EJB level, the clustered JNDI cmi provides Load Balancing / High Availability.

The database is shared by the JOnAS servers.

The versions assumed here are: Apache 2.0, JOnAS 3.1.x/Tomcat 4.1.x package.

The architecture presented in this document is shown in the following illustration:



This architecture provides:

- **Load balancing**: Requests can be dispatched over a set of servers to distribute the load. This improves the "scalability" by allowing more requests to be processed concurrently.
- **High Availability (HA)**: having several servers able to fulfill a request makes it possible to ensure that, if a server dies, the request can be sent to an available server (thus the load−balancing algorithm ensures that the server to which the request will be sent is available). Therefore, "Service Availability" is achieved.
- **Failover at Servlet / JSP Level**: This feature ensures that, if one JSP/servlet server goes down, another server is able to transparently take over, i.e. the request will be switched to another server without service disruption. This means that it will not be necessary to start over, thus achieving Continuity.

However, failover at EJB level is not available. This means that no State Replication is provided. The mechanism to provide failover at EJB level is under development and will be available in a coming version of JOnAS.

Products Installation

This chapter provides information about installing Apache and JOnAS / Tomcat. The versions assumed here are: Apache 2.0 and the package JOnAS 3.1.x /Tomcat 4.1.x.

Installing Apache

1. Download Apache HTTP server source code from the Apache site.
2. Extract the source code.
   ```
   gunzip httpd-2_0_XX.tar.gz
   tar xvf httpd-2_0_XX.tar
   ```
3. Compile and Install.
   ```
   ./configure
   make
   make install
   ```

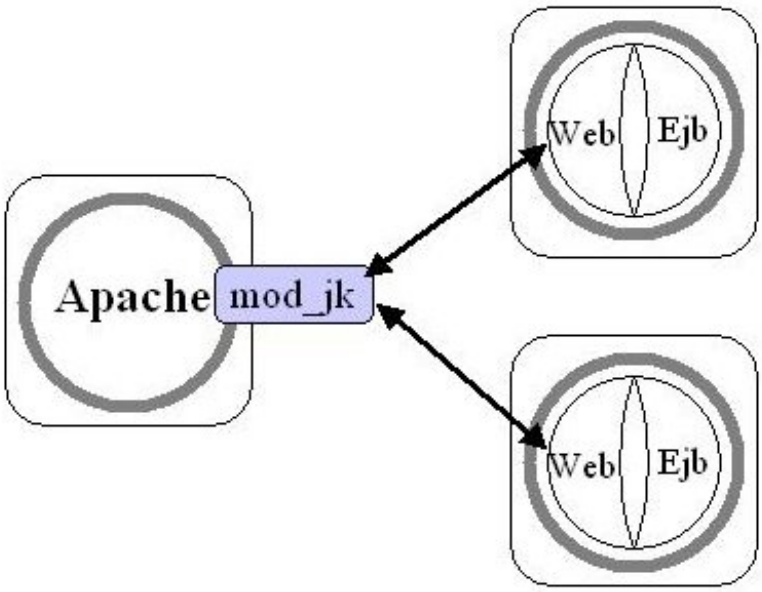
A binary version is also available for installation at the Apache site.

Installing the package JOnAS / Tomcat

Refer to Installing JOnAS with a web container from scratch.

Load balancing at web level with mod_jk

This chapter describes how to configure Apache, Tomcat, and JOnAS to run the architecture shown in the following illustration:



Configuring the JK Module

JK module principles

Mod_jk is a plug−in that handles the communication between Apache and Tomcat.
Mod_jk uses the concept of worker. A worker is a Tomcat instance that is running to perform servlet requests coming from the web server. Each worker is identified to the web server by the host on which it is located, the port where it listens, and the communication protocol used to exchange messages. In this configuration there is one worker for each Tomcat instance and one worker that will handle the load balancing (this is a specific worker with no host and no port number). All workers are defined in a file called worker.properties.

Note: this module can also be used for site partitioning.

install Mod_jk

The easiest way to obtain this plug−in is to download the binary from the Tomcat Site and place it in the directory libexec (for unix) or modules (for windows or Mandrake) of the Apache installation directory.

Configure Apache

- httpd.conf
  Create a file tomcat_jk.conf, which must be included in $APACHE_HOME/conf/httpd.conf.
  This file should load the module mod_jk:

```
LoadModule jk_module modules/mod_jk.so (for windows)
LoadModule jk_module libexec/mod_jk.so (for Unix)
AddModule mod_jk.c
```

  And configure mod_jk:

```
# Location of the worker file
JkWorkersFile "/etc/httpd/conf/jk/workers.properties"
# Location of the log file
JkLogFile "/etc/httpd/jk/logs/mod_jk.log"
# Log level : debug, info, error or emerg
JkLogLevel emerg
# Assign specific URL to Tomcat workers
JkMount /admin loadbalancer
JkMount /admin/* loadbalancer
JkMount /examples loadbalancer
JkMount /examples/* loadbalancer
```

- worker.properties
  This file should contain the list of workers first:
  `worker.list=<a comma separated list of worker names>`

then the properties of each worker:
`worker.<worker name>.<property>=<property value>`

The following is an example of a worker.properties file:

```
# List the workers name
worker.list=worker1,worker2,loadbalancer
# ----------------
# First worker
# ----------------
worker.worker1.port=8009
worker.worker1.host=server1
worker.worker1.type=ajp13
# Load balance factor
worker.worker1.lbfactor=1
# ----------------
# Second worker
# ----------------
worker.worker2.port=8009
worker.worker2.host=server2
worker.worker2.type=ajp13
worker.worker2.lbfactor=1
# ----------------------
# Load Balancer worker
# ----------------------
worker.loadbalancer.type=lb
worker.loadbalancer.balanced_workers=worker1,worker2
```

Configure Tomcat

To configure Tomcat, perform the following configuration steps for each Tomcat server.

1. Configure Tomcat for the connector AJP13. In the file conf/server.xml of the JOnAS installation directory, add (if not already there):

   ```
   <!-- Define an AJP 1.3 Connector on port 8009 -->
   <Connector className="org.apache.ajp.tomcat4.Ajp13Connector"
       port="8009" minProcessors="5" maxProcessors="75"
       acceptCount="10" debug="20"/>
   ```

2. Define the jvmRoute.
   In the file conf/server.xml of the JOnAS installation directory, add a unique route to the Catalina engine.
   Replace the line:
   `<Engine name="Standalone" defaultHost="localhost" debug="0">`
   with:

```
<Engine jvmRoute="worker1" name="Standalone" defaultHost="localhost"
debug="0">
```

Note: The jvmRoute name should be the same as the name of the associated worker defined in worker.properties. This will ensure the Session affinity.

Configuring JOnAS

In the JOnAS−specific deployment descriptor, add the tag `shared` for the entity beans involved and set it to `true`. When this flag is set to true, multiple instances of the same entity bean in different JOnAS servers can access a common database concurrently.

The following is an example of a deployment descriptor with the flag `shared`:

```
<jonas−ejb−jar>
  <jonas−entity>
    <ejb−name>Id_1</ejb−name>
    <jndi−name>clusterId_1</jndi−name>
    <shared>true</shared>
    <jdbc−mapping>
      <jndi−name>jdbc_1</jndi−name>
      <jdbc−table−name>clusterIdentityEC</jdbc−table−name>
      <cmp−field−jdbc−mapping>
        <field−name>name</field−name>
        <jdbc−field−name>c_name</jdbc−field−name>
      </cmp−field−jdbc−mapping>
      <cmp−field−jdbc−mapping>
        <field−name>number</field−name>
        <jdbc−field−name>c_number</jdbc−field−name>
      </cmp−field−jdbc−mapping>
      <finder−method−jdbc−mapping>
        <jonas−method>
          <method−name>findByNumber</method−name>
        </jonas−method>
        <jdbc−where−clause>where c_number = ?</jdbc−where−clause>
      </finder−method−jdbc−mapping>
      <finder−method−jdbc−mapping>
        <jonas−method>
          <method−name>findAll</method−name>
        </jonas−method>
        <jdbc−where−clause></jdbc−where−clause>
      </finder−method−jdbc−mapping>
    </jdbc−mapping>
  </jonas−entity>
</jonas−ejb−jar>
```

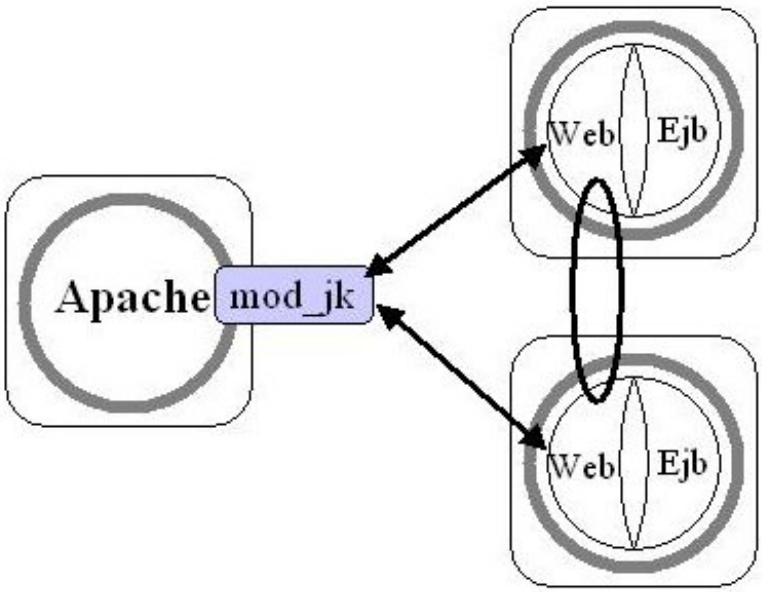
Running a Web Application

The web application is now ready to run:

1. Start the jonas servers: `jonas start`.
2. Restart Apache: `/usr/local/apache2/bin/apachectl restart`.
3. Use a browser to access the welcome page, usually index.html.

Session Replication

The intent of this chapter is to configure Apache, Tomcat, and JOnAS to run the following architecture:



The term session replication is used when the current service state is being replicated across multiple application instances. Session replication occurs when the information stored in an HttpSession is replicated from, in this example, one servlet engine instance to another. This could be data such as items contained in a shopping cart or information being entered on an insurance application. Anything being stored in the session must be replicated for the service to failover without a disruption.

The solution chosen for achieving Session replication is called in−memory−session−replication. It uses a group communication protocol written entirely in Java, called JavaGroups. JavaGroups is a communication protocol based on the concept of virtual synchrony and probabilistic broadcasting.

The follow describes the steps for achieving Session replication with JOnAS.

- The mod_jk is used to illustrate the Session Replication. Therefore, first perform the configuration steps presented in the chapter Load Balancing at Web level with mod_jk.

- On the JOnAS servers, open the `<JONAS_BASE>/conf/server.xml` file and configure the `<context>` as described:

```
<Context path="/replication−example" docBase="replication−example" debug="99"
                    reloadable="true" crossContext="true"
                    className="org.objectweb.jonas.web.catalina41.JOnASStandardContext">
               <Logger className="org.apache.catalina.logger.FileLogger"
                        prefix="localhost_replication_log." suffix=".txt"
                        timestamp="true"/>
                <Valve className="org.apache.catalina.session.ReplicationValve"
                         filter=".*\.gif;.*\.jpg;.*\.jpeg;.*\.js" debug="0"/>
                <Manager className="org.apache.catalina.session.InMemoryReplicationManager"
                            debug="10"
                            printToScreen="true"
                            saveOnRestart="false"
                            maxActiveSessions="−1"
                            minIdleSwap="−1"
                            maxIdleSwap="−1"
                            maxIdleBackup="−1"
                            pathname="null"
                            printSessionInfo="true"
                            checkInterval="10"
                            expireSessionsOnShutdown="false"
                            serviceclass="org.apache.catalina.cluster.mcast.McastService"
                            mcastAddr="237.0.0.1"
                            mcastPort="45566"
                            mcastFrequency="500"
                            mcastDropTime="5000"
                            tcpListenAddress="auto"
                            tcpListenPort="4001"
                            tcpSelectorTimeout="100"
                            tcpThreadCount="2"
                            useDirtyFlag="true">
               </Manager>
</Context>
```

  Note: The multicast address and port must be identically configured for all JOnAS/Tomcat instances.

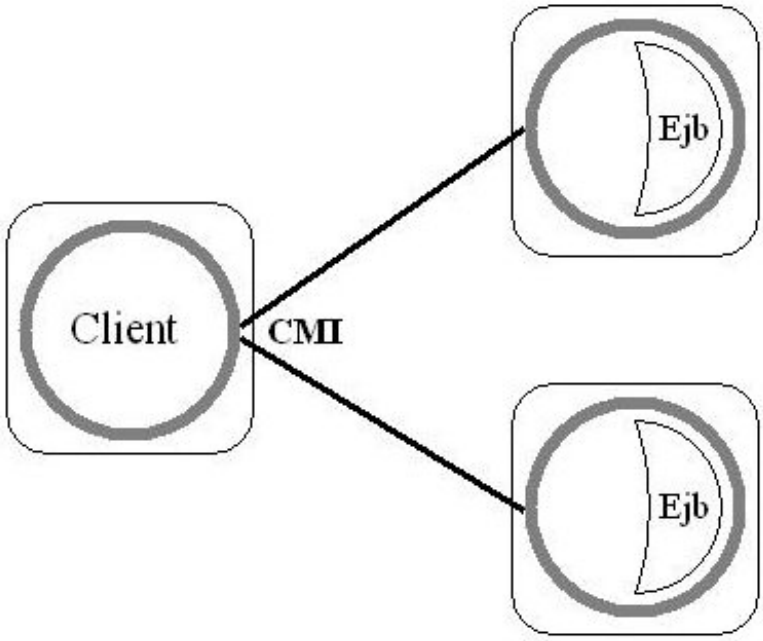
Running your Web Application

The web application is now ready to run in the cluster:

1. Start the JOnAS servers : `jonas start`.
2. Restart Apache : `/usr/local/apache2/bin/apachectl restart`.
3. Use a browser to access the welcome page, usually index.html.

Load Balancing at EJB level

The intent of this chapter is to configure JOnAS to run the following architecture:



CMI Principles

CMI is a new ORB used by JOnAS to provide clustering for load balancing and high availability. Several instances of JOnAS can be started together in a cluster to share their EJBs. It is possible to start the same EJB on each JOnAS, or to distribute their load. A URL referencing several JOnAS instances can be provided to the clients. At lookup time, a client randomly chooses one of the available servers to request the required bean. Each JOnAS instance has the knowledge (through Javagroups) of the distribution of the Beans in the cluster. An answer to a lookup is a special clustered stub, containing stubs to each instance known in the cluster. Each method call on the Home of the bean can be issued by the stub to a new instance, to balance the load on the cluster. The default algorithm used for load distribution is currently a weighted round robin.

CMI Configuration

- In the build.properties of the application, set the protocol name to cmi **before compilation**:
  protocols.names=cmi
- In the file carol.properties of the directory $JONAS_BASE/conf, set the protocol to cmi:
  carol.protocols=cmi
- In the file carol.properties, configure the multicast address, the group name, the round−robin weight factor, etc.
  The following is a configuration example:

 # java.naming.provider.url property
carol.cmi.url=cmi://localhost:2002

 # Multicast address used by the registries in the cluster
carol.cmi.multicast.address=224.0.0.35:35467

 # Groupname for Javagroups
carol.cmi.multicast.groupname=G1

 # Factor used for this server in weighted round robin algorithms
carol.cmi.rr.factor=100

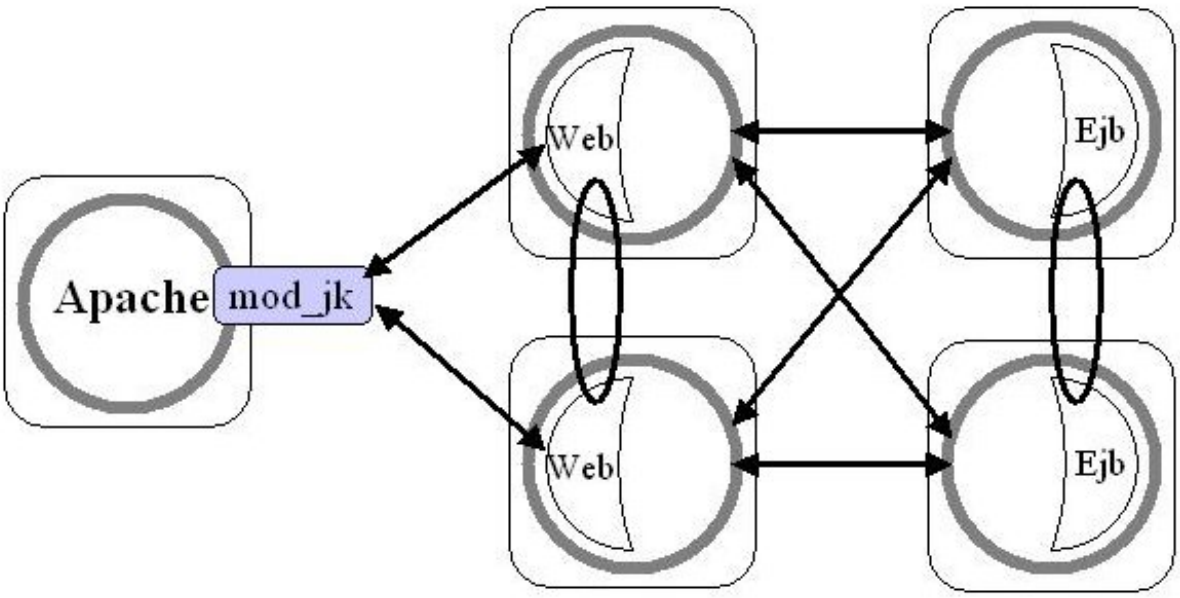
- For the client, specify the list of registries available in the carol.properties file:
  carol.cmi.url=cmi://server1:port1[,server2:port2...]

Note 1: The multicast address and group name must be the same for all JOnAS servers in the cluster.

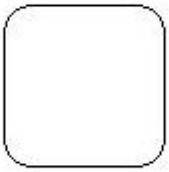
Note 2: If Tomcat Replication associated to cmi is used, the multicast addresses of the two configurations must be different.

Preview of a coming version

A solution that enables failover at EJB level is currently under development. This signifies state replication for stateful session beans and entity beans.
This will enable the following architecture:



Used symbols

	A node (computer) that hosts one or more servers		
	A web container		An ejb container
	A JOnAS instance that hosts a web container		A JOnAS instance that hosts an ejb container
	A JOnAS instance that hosts a web container and an ejb container		
	An Apache server with the mod_jk module		

References

- Working with mod_jk
- Tomcat workers Howto
- Apache JServ Protocol version 1.3 (ajp13)
- Apache − Tomcat HOWTO
- Apache 1.3.23 + Tomcat 4.0.2 + Load Balancing
- Tomcat 4 Clustering

Howto: How to use AXIS in JOnAS

This guide describes basic Axis use within JOnAS. It assumes that the reader does not require any explanation about Axis−specific tasks (e.g., axis deployment with **WSDD**). Before deployment in Axis, the user must verify that the deploy.wsdd file **matches the site machine configuration** (jndiURL parameter in particular:<*parameter name="jndiURL" value="rmi://localhost:1099"/*>).

This document describes two ways to make an EJB (stateless SB) available as a Web Service with JOnAS:

1. Axis runs in a unique Webapp, the **stateless SB** (Session Bean) is packaged in a separate ejb−jar (or even EAR). The intent of this approach is to make EJBs from differents packages that are **already deployed** accessible as Web Services via a single Axis Webapp deployment. The drawback is that Web Services are **centralized** in one Webapp only and the only way to distinguish between them for access is by the <service−name>, not by the <context−root>/<service−name>. In addition, the ejb−jar files that contain the Web Services must be **included in the Webapp**.
2. The accessed EJB(s) are packaged with the Axis Webapp in an EAR archive. With this approach, **the ejb−jar files do not have to be included** in the Webapp WEB−INF/lib directory; different Applications that contain Web Services can be hosted, providing the capability of distinguishing between Web Services of different applications.

Libraries

JOnAS incorporates all the necessary libraries, including:

- **JAX−R**: Reference Implementation from Sun
- **JAX−M**: Reference Implementation from Sun
- **JAX−P**: Xerces XML parser (version 2.4.0)
- **AXIS**: Soap implementation from Apache (with all dependent libs : jaxrpc.jar, ...)

(**JAX−M** and **JAX−R** are parts of the Web Services Development Pack from Sun.)

1. Unique Axis Webapp

Constraints:

- The EJBs exposed as WebServices must have **remote interfaces**.
- The Axis Webapp must have in its **WEB−INF/lib directory all the ejb−jar files** containing Beans exposed as Web Services.

Usage:

- **Deploy** the ejb−jars or EARs containing Web Services.
- **Deploy** the Axis Webapp (containing the **ejb−jar** files).

- Use the AdminClient tool to deploy the Web Services (with a **.wsdd** file).
- Example: *jclient org.apache.axis.client.AdminClient −hjonasServerHostname −p8080 deploy.wsdd*

Example: Refer to the *separate_axis* example (in the $JONAS_ROOT/examples directory).

2. Embedded Axis Webapp

Constraints:

- The EJBs exposed as Web Services can have either **local or remote interfaces**.
- The EAR must contain a Webapp including a **web.xml with Axis servlet mapping**.

Usage:

- **Deploy** the application archive (EAR) :
- Use the AdminClient tool to deploy the webservices (with a **.wsdd** file)
- Example: *jclient org.apache.axis.client.AdminClient −lhttp://localhost:8080/hello/servlet/AxisServlet deploy.wsdd*
- *Be careful to use a good URL to reach the AxisServlet.*

Example: Refer to the *embedded_axis* example (in the $JONAS_ROOT/examples directory).

3. Tests

When everything is deployed and running, use the following URL to view the deployed Web Services:
http://<yourserver>:<port>/<yourwebapp>/servlet/AxisServlet
This page will display a link for each Web Service with the **WSDL file** (automatically **generated by Axis** from the Java Interfaces).
Use the following URL to access your Web Service (add ?WSDL for the associated WSDL file):
http://<yourserver>:<port>/<yourwebapp>/services/<Service−Name>

A client class can now be run against the Web Service. Note that any language (with Web Services capabilities) can be used for the client (C#, Java, etc.).

Tools:

Use jclient to deploy your Web Services (in the Axis way):
jclient org.apache.axis.client.AdminClient [OPTIONS] <WSDD−file>
[OPTIONS] :
−l<*URL*>: the **location of the AxisServlet** servlet (default : http://localhost:8080/axis/servlet/AxisServlet)
−p<*port*>: the **port** of the listening http daemon (default : 8080)

−**h**<*host*>: the **hostname** of the server running the JOnAS server (default : localhost)

Howto: Using WebSphere MQ JMS guide

This document explains how WebSphere MQ (formerly MQSeries) may be used as JMS provider within a JOnAS application server.

WebSphere MQ is the messaging platform developed by IBM. It provides Java and JMS interfaces. Documentation may be found at http://www−3.ibm.com/software/integration/mqfamily/library/manualsa/.

The document has been written after an integration work done with JOnAS 3.3 and 3.3.1 and WebSphere MQ 5.3.

The content of this guide is the following:

Architectural rules

WebSphere MQ, contrary to JORAM or SwiftMQ, can not run collocated with JOnAS. WebSphere MQ is an external software which needs to be independently administered and configured.

Administering WebSphere MQ consists in:

- Creating and configuring resources (such as queues) through the WebSphere MQ Explorer tool.
- Creating the corresponding JMS objects (`javax.jms.Queue`, `javax.jms.Topic`, `javax.jms.QueueConnectionFactory`, etc.), and binding them to a registry.

The link between JOnAS and WebSphere MQ will be established thanks to the registry. WebSphere MQ JMS objects will be bound to the JOnAS registry. JMS lookups will then return the WebSphere MQ JMS objects, and messaging will take place through these objects.

Given the complex configuration of WebSphere MQ JMS objects, it is not possible to create them from JOnAS. Thus, during the starting phase, a JOnAS server expects WebSphere MQ JMS objects to have already been bound to the registry. Obviously there is a need to start an independent registry, to which WebSphere MQ may bind its JMS objects, and which may also be used by the starting JOnAS server. The start−up sequence looks as follows:

1. Starting a registry.
2. Creating and binding WebSphere MQ JMS objects.
3. Launching the JOnAS server.

The following architecture is proposed:

- A JOnAS server (for example called "Registry") only providing a registry.
- A JOnAS server (for example called "EJB") using the registry service of server "Registry".
- Plus, of course, a WebSphere MQ server running locally.

Setting the JOnAS environment

The proposed architecture requires to run two JOnAS server instances. In order to do that, the following steps are proposed:

1. Create two base directories. For example `JONAS_REGISTRY` and `JONAS_EJB`.
2. Set the JONAS_BASE environment variable so that it points towards the `JONAS_REGISTRY` directory.
3. In the $JONAS_ROOT directory, type: *ant create_jonasbase*
4. Set the JONAS_BASE environment variable so that it points towards the `JONAS_EJB` directory.
5. In the $JONAS_ROOT directory, type: *ant create_jonasbase*

The JOnAS servers may now be configured independently.

Configuring the "Registry" server

The "Registry" server is the JOnAS server which will host the registry service. Its configuration files are in `JONAS_REGISTRY/conf`.

In the `jonas.properties` files, only declare the registry and jmx services:

```
jonas.services    registry,jmx
```

In the carol.properties file, declare the jeremie protocol:

```
carol.protocols=jeremie
```

You may also configure its port:

```
carol.jeremie.url=jrmi://localhost:2000
```

Configuring the "EJB" server

The "EJB" server is the JOnAS server which will be used as the application server. Its configuration files are in `JONAS_EJB/conf`. Libraries must be added in `JONAS_EJB/lib/ext`.

In the jonas.properties files, set the registry service as remote:

```
jonas.service.registry.mode    remote
```

... and the JMS service as WebSphere MQ:

```
jonas.service.jms.mom    org.objectweb.jonas_jms.JmsAdminForWSMQ
```

In the carol.properties file, declare the jeremie protocol and set the correct port:

```
carol.protocols=jeremie
carol.jeremie.url=jrmi://localhost:2000
```

In `lib/ext`, the following libraries must be added:

- `com.ibm.mqjms.jar`, including WebSphere MQ JMS classes.
- `com.ibm.mq.jar`, also a WebSphere MQ library.

Configuring WebSphere MQ

WebSphere MQ JMS administration is documented in chapter 5 of the "WebSphere MQ Using Java" document.

The configuration file of the JMS administration tool must be edited so that the JOnAS registry is used for binding the JMS objects. This file is the `JMSAdmin.config` file located in WebSphereMQ's `Java/bin` directory. Set the factory and provider URL as follows:

```
INITIAL_CONTEXT_FACTORY=org.objectweb.jeremie.libs.services.registry.jndi.JRMIInitial
PROVIDER_URL=jrmi://localhost:2000
```

You also need to add JOnAS's `client.jar` library to WebSphere MQ's classpath.

When starting, JOnAS expects JMS objects to have been created and bound to the registry. Those objects are connection factories, needed for connecting to WebSphere MQ destinations, and destinations.

JOnAS automatically tries to access the following factories:

- An `XAConnectionFactory`, bound with name "wsmqXACF".
- An `XAQueueConnectionFactory`, bound with name "wsmqXAQCF".
- An `XATopicConnectionFactory`, bound with name "wsmqXATCF".
- A `ConnectionFactory`, bound with name "JCF".
- A `QueueConnectionFactory`, bound with name "JQCF".
- A `TopicConnectionFactory`, bound with name "JTCF".

It one of these objects can't be found, JOnAS will simply print a message which looks as follows:

JmsAdminForWSMQ.start : WebSphere MQ XAConnectionFactory could not be retrieved from JNDI

This does not prevent JOnAS from working. However if no connection factory at all is available, no JMS operations

will be possible from JOnAS.

If destinations have been declared in the `jonas.properties` file, JOnAS will also expect to find them. For example, if the following destinations are declared:

```
jonas.service.jms.topics    sampleTopic
jonas.service.jms.queues    sampleQueue
```

The server expects to find the following JMS objects in the registry:

- A `Queue`, bound with name "sampleQueue".
- A `Topic`, bound with name "sampleTopic".

If one of the declared destination can't be retrieved, the following message appears, and the server stops:

JOnAS error: org.objectweb.jonas.service.ServiceException : Cannot init/start service jms': org.objectweb.jonas.service.ServiceException : JMS Service Cannot create administered object: java.lang.Exception: WebSphere MQ Queue creation impossible from JOnAS

Contrary to connection factories, the JOnAS administration tool allows to create destinations. Since it is not possible to create WebSphere MQ JMS objects from JOnAS, this will work only if the destinations are preliminary created and bound to the registry.

For example, if you try to create through the JonasAdmin tool a queue named "myQueue", this will only work if:

- You have created a queue through the WebSphere MQ Explorer tool.
- You have created the corresponding JMS `Queue` and bound it to the registry with the name "myQueue".

Launching WebSphere MQ administration tool is simply done by typing: *JMSAdmin*

The following prompt appears: *InitCtx>*

Creating a `QueueConnectionFactory`, and binding it with name "JQCF" is done by typing:

InitCtx> DEF QCF(JQCF)

More parameters may be put (for example for specifying the queue manager).

Creating a `Queue`, representing a WebSphere MQ queue named "myWSMQqueue", and binding it with name "sampleQueue" is done by typing:

InitCtx> DEF Q(sampleQueue) QUEUE(myWSMQqueue)

Objects bound in the registry may be seen by typing:

InitCtx> DIS CTX

Starting the application

Starting the registry server:

1. Clean the local CLASSPATH: *set/export CLASSPATH=""*
2. Set the JONAS_BASE variable so that it points towards `JONAS_REGISTRY`.
3. Start the JOnAS server: *jonas start −n Registry*

Administering WebSphere MQ:

1. In WebSphere MQ's `Java/bin` directory, launch the JMSAdmin tool: *JMSAdmin*
2. Create the needed JMS objects.

Starting the EJB server:

1. Clean the local CLASSPATH: *set/export CLASSPATH=""*
2. Set the JONAS_BASE variable so that it points towards `JONAS_EJB`.
3. Start the JOnAS server: *jonas start −n EJB*

Starting an EJB client:

1. Add in the jclient classpath the `ibm.com.mq.jar` and `ibm.com.mqjms.jar` libraries.
2. Launch the client: *jclient ...*

Limitations

Using WebSphere MQ as JMS transport within JOnAS raises some limitations compared to using JORAM or SwiftMQ.

First of all, WebSphere MQ is compliant with the old 1.0.2b JMS specifications. Code written following the JMS 1.1 latest spec (such as the jms samples provided with JOnAS) won't work with WebSphere MQ.

Depending on the WebSphere MQ distribution, JMS Publish/Subscribe may not be available. In this case, the message−driven bean samples provided with JOnAS won't work. That's why a specific sample is provided.

Finally, for an unknown reason, asynchronous consumption of messages (through message driven beans) does not work in transactional mode. Further inquiry is needed to solve this issue.

Howto: Web Service Interoperability between JOnAS and Weblogic

This guide describes the basic use of web services between JOnAS and Weblogic server. It assumes that the reader does not require any explanation about Axis−specific tasks (axis deployment with **WSDD**, etc.).

This document describes the following two aspects:

1. Access a web service deployed on JOnAS from an EJB deployed on Weblogic server.
2. Access a web service deployed on Weblogic server from an EJB deployed on JOnAS.

Libraries

JOnAS incorporates all the necessary libraries, including:

- **JAX−R**: Reference Implementation from Sun
- **JAX−M**: Reference Implementation from Sun
- **JAX−P**: Xerces XML parser (version 2.4.0)
- **AXIS**: Soap implementation from Apache (with all dependent libs: jaxrpc.jar, etc.)

(**JAX−M** and **JAX−R** are parts of the Web Services Development Pack from Sun.)

Weblogic incorporates all the necessary libraries, including:

- **All libraries for using webservice are contained in webserviceclient.jar.**

Access a web service deployed on JOnAS from an EJB deployed on Weblogic server

Web Service Development on JOnAS

Also refer to the document How to use Axis with JOnAS, which describes how to develop and deploy web services on JOnAS.

EJB Creation on JOnAS

To create a web service based on an EJB, first create a stateless EJB.
Then, create a web application (.war) or an application (.ear) with this EJB that will define a URL with access to the Web Service.

WebService Deployment Descriptor (WSDD)

This section describes the deployment descriptor of the web service.
To deploy a web service based on an EJB, specify the various elements in the WSDD.
This WSDD enables the web service to be mapped on an EJB, by specifying the different EJB classes used.

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

<!-- AXIS deployment file for HelloBeanService -->
     <service name="WebServiceName" provider="java:EJB">

<!-- JNDI name specified in jonas-EJB.xml -->
      <parameter name="beanJndiName"   value="EJB_JNDI_Name"/>

<!-- use of remote interfaces to access the EJB is allowed, but this example
uses local interfaces -->
          <parameter name="homeInterfaceName"   value="EJB_Home"/>
          <parameter name="remoteInterfaceName"  value="EJB_Interface"/>

<!-- Specify here allowed methods for Web Service access (* for all) -->
        <parameter name="allowedMethods"   value="*"/>

    </service>
</deployment>
```

The various tags allow mapping of the web service on different java classes.
If a web service uses a complex type, this complex type must be mapped with a java class. To do this, two tags can be used:

```
<beanMapping qName="ns:local" xmlns:ns="someNameSpace"
languageSpecificType="java:my.class"/>
```

This maps the QName [someNameSpace]:[local] with the class my.class.

```
<typeMapping qname="ns:local" wmlns:ns="someNamespace"
languageSpecificType="java:my.class"
serializer="my.java.SerializerFactory"
deserializer="my.java.DeserializerFactory"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
```

Where QName [someNamespace]:[local] is mapped with my.class. The serializer used is the class my.java.SerializerFactory, and the deserializer used is my.java.DeserializerFactory.

Web Service Deployment on JOnAS

First, deploy the web application or the application containing the EJB.
Then, deploy the web service using the Axis client tool: *jclient org.apache.axis.client.AdminClient −hjonasServerHostname −p8080 deploy.wsdd*
Note: from JOnAS 3.3, jclient no more include WebServices libraries in the CLASSPATH. So you have to manually add this jar in your CLASSPATH:
export CLASSPATH=$CLASSPATH:$JONAS_ROOT/lib/webservices_axis.jar

The web service WSDL is accessible from the url: http://<host>:<port>/<url−servlet>/<webservicename>?wsdl.

EJB proxy development for Weblogic Server

This EJB provides access to the web service deployed on JOnAS from Weblogic server.

Generation of web service client class

To access the web service, generate the client class using the ant task clientgen.
For example:

<clientgen wsdl="<wsdl_url>" packageName="my.package" clientJar="client.jar" generatePublicFields="True" keepGenerated="True"/>

This command creates four classes:

- Service implementation
- Java Interface
- Stub class
- Service interface corresponding web service

The tool can also generate the java classes corresponding to future complex types of web service.

Build the EJB

Then, call the web service in the EJB proxy code using these generated classes.
For example:

```
try {
       WSNAME_Impl tsl=new WSNAME_Impl(); // access web service impl
       EJB_endpoint tsp = tsl.getEJB_endpoint(); // access WS endpoint interface
       ComplexType tr=tsp.method(param);
    }  catch (Exception e) {
       e.printStackTrace(System.err);
    };
```

Deploy the EJB on Weblogic Server

Deploy this EJB using the weblogic administration console.

Access a web service deployed on Weblogic server from an EJB deployed on JOnAS

Web Service Development for Weblogic Server

Creation of an application

To create a web service, first develop the corresponding EJB application. Compile the EJB classes and create a jar file. To create the EJB's container, apply the ant task `wlappc` to the jar file. For example: `<wlappc debug="${debug}" source="interface_ws_jonas.jar" classpath="{java.class.path}:interface_ws_jonas.jar"`
Then, use the ant task servicegen to create the ear application containing the web service.

```
<servicegen
  destEar="ears/myWebService.ear"
  contextURI="web_services" >
  <service
    ejbJar="jars/myEJB.jar"
    targetNamespace="http://www.bea.com/examples/Trader"
    serviceName="TraderService"
    serviceURI="/TraderService"
    generateTypes="True"
    expandMethods="True" >
  </service>
 </servicegen>
```

THE ANT USED IS PROVIDED BY WEBLOGIC

WebService Deployment

Deploy the webservice using the Weblogic administration console, and deploy the corresponding application.
The WSDL is accessible at http://<host>:<port>/webservice/web_services?WSDL.

EJB proxy development for JOnAS

This EJB provides access to the web service deployed on Weblogic from JOnAS.

Generation of web service client class

To access a web service, generate a client class using the axis tool WSDL2Java <webservice−url−wsdl>.
This command creates four classes:

- ${WSNAME}Locator.java: Service implementation
- ${WSNAME}Port.java: Java Interface
- ${WSNAME}PortStub.java: Stub class
- ${WSNAME}.java: Service interface corresponding web service

The tool also generates the java class corresponding to future complex types of web service.

Build the EJB

Then, use this generated class to call the web service in the EJB proxy code.
For example:

```
try {
    WSNAMELocator tsl=new WSNAMELocator();
    WSNAMEPort tsp = tsl.getWSNAMEPort();
    ComplexType tr=tsp.method(param);
    ...
    } catch (Exception e) {
         e.printStackTrace(System.err);
    };
```

Deploy the EJB on JOnAS

Deploy the EJB using the JOnAS administration console or command.

Howto: RMI−IIOP interoperability between JOnAS and Weblogic

This guide describes the basic interoperability between JOnAS and Weblogic Server using RMI−IIOP (the examples in this document assume that the Sun rmi/iiop of the JDK is used).

The content of this guide is the following:

1. Accessing an EJB deployed on JOnAS from an EJB deployed on Weblogic server using RMI−IIOP.
2. Accessing an EJB deployed on Weblogic Server from an EJB deployed on JOnAS using RMI−IIOP.

Accessing an EJB deployed on JOnAS from an EJB deployed on Weblogic server using RMI−IIOP

JOnAS Configuration

No modification to the EJB code is necessary. However, to deploy it for use with the iiop protocol, add the tag protocols and indicate iiop when creating the build.xml.
For example:

```
<jonas destdir="${dist.ejbjars.dir}" classpath="${classpath}" jonasroot="${jonas.root}" protocols="
```

If GenIC is being used for deployment, the `-protocols` option can be used. Note also that an EJB can be deployed for several protocols. For more details about configuring the communication protocol, refer to the JOnAS Configuration Guide.

For the JOnAS server to use RMI−IIOP, the JOnAS configuration requires modification. The iiop protocol must be selected in the
file carol.properties. Refer also to the JOnAS Configuration Guide for details about configuring the communication protocol.
This modification will allow an EJB to be created using the RMI−IIOP protocol.

EJB proxy on Weblogic

To call an EJB deployed on JOnAS that is accessible through RMI−IIOP, load the class com.sun.jndi.cosnaming.CNCtxFactory as the initial context factory.
In addition, specify the JNDI url of the server name containing the EJB to call: "iiop://<server>:port."
For example:

```
try {
```

```
  Properties h = new Properties();
  h.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.cosnaming.CNCtxFactory");
  h.put(Context.PROVIDER_URL, "iiop://<server>:<port>");
  ctx=new InitialContext(h);
  }
catch (Exception e) {
  ...
  }
```

Then, the JOnAS EJB is accessed in the standard way.

Access an EJB deployed on Weblogic Server by an EJB deployed on JOnAS using RMI−IIOP

Weblogic Configuration

No modification to the EJB code is necessary. However, to deploy the EJB for use with the iiop protocol, add the element iiop="true" on the wlappc task when creating the build.xml.
For example:

```
wlappc debug="${debug}" source="ejb.jar" iiop="true" classpath="${class.path}"
```

EJB proxy on JOnAS

To call an EJB deployed on Weblogic Server that is accessible through RMI−IIOP, specify the JNDI url of the server name containing the EJB to call. This url is of type: "iiop://<server>:port."
For example:

```
try {
   Properties h = new Properties();
   h.put(Context.PROVIDER_URL, "iiop://<server>:<port>");
   ctx=new InitialContext(h);
}
catch (Exception e) {
    ...
}
```

Then, the EJB deployed on Weblogic server is accessed in the standard way.

Howto: Interoperability between JOnAS and CORBA

This guide describes the basic interoperability between JOnAS and CORBA using RMI−IIOP (the examples in this document assume that the Sun rmi/iiop of the JDK 1.4 is used).

The content of this guide is the following:

Accessing an EJB deployed a on JOnAS server by a CORBA client

JOnAS Configuration

No modification to the EJB code is necessary. However, the EJB should be deployed for the iiop protocol (e.g. when the build.xml is created, add the tag "protocols" and specify "iiop").
For example:

```
<jonas destdir="${dist.ejbjars.dir}" classpath="${classpath}" jonasroot="${jonas.root}" protocols="
```

If GenIC is used for deployment, the `-protocols` option can be used. Note also that an EJB can be deployed for several protocols. Refer to the JOnAS Configuration Guide for more details about configuring the communication protocol.

The JOnAS configuration must be modified for the JOnAS server to use RMI−IIOP.
Choose the iiop protocol in the file carol.properties. Refer also to the JOnAS Configuration Guide for details about configuring the communication protocol.
These modifications will make it possible to create an EJB using the RMI−IIOP protocol.

RMIC to create IDL files used by the Corba Client

To call an EJB deployed on JOnAS that is accessible through RMI−IIOP, use the rmic tool on the EJB Remote interface and EJB Home interface to create the idl files. Example: `rmic −classpath $JONAS_ROOT/lib/common/j2ee/ejb.jar −idl package1.Hello`

This action generates several idl files:

```
package1/Hello.idl
package1/HelloHome.idl

java/io/FilterOutputStream.idl
java/io/IOException.idl
java/io/IOEx.idl
java/io/OutputStream.idl
```

```
java/io/PrintStream.idl
java/io/Writer.idl
java/io/PrintWriter.idl

java/lang/Exception.idl
java/lang/Ex.idl
java/lang/Object.idl
java/lang/StackTraceElement.idl
java/lang/ThrowableEx.idl
java/lang/Throwable.idl

javax/ejb/EJBHome.idl
javax/ejb/EJBMetaData.idl
javax/ejb/EJBObject.idl
javax/ejb/Handle.idl
javax/ejb/HomeHandle.idl
javax/ejb/RemoveException.idl
javax/ejb/RemoveEx.idl

org/omg/boxedRMI/seq1_octet.idl
org/omg/boxedRMI/seq1_wchar.idl

org/javax/rmi/CORBA/ClassDesc.idl
org/omg/boxedRMI/java/lang/seq1_StackTraceElement.idl
```

Copy these files to the directory in which CORBA client development is being done.

CORBA Client Development

1. idlj

Once idl files are generated, apply the idlj tool to build java files corresponding to the idl files (idlj = idl to java). To do this, apply the idlj tool to the Remote interface idl file and the Home interface idl file. Example: `idlj −fclient −emitAll package1/Hello.idl`

The idlj tool also generates bugged classes. Be sure to put the _read and _write method in comment in the class _Exception.java, CreateException.java, RemoveException.java.

Additionally, the class OutputStream.java, PrintStream.java, PrintWriter.java, Writer.java, FilterOuputStream.java must extend Serializable and then replace

```
 ((org.omg.CORBA_2_3.portable.OutputStream) ostream).write_value(value,id());
```

with

```
 ((org.omg.CORBA_2_3.portable.OutputStream) ostream).write_value((Serializable) value,id());
```

in the write method.

2. Client

Create the Corba client.

```
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class Client {
    public static void main(String args[]) {
        try {
            //Create and initialize the ORB
            ORB orb=ORB.init(args,null);
            //Get the root naming context
            org.omg.CORBA.Object objRef=orb.resolve_initial_references("NameService");
            NamingContext ncRef= NamingContextHelper.narrow(objRef);

            //Resolve the object reference in naming
            //make sure there are no spaces between ""
            NameComponent nc= new NameComponent("HelloHome","");
            NameComponent path[] = {nc};
            HelloHome tradeRef=HelloHomeHelper.narrow(ncRef.resolve(path));

            //Call the Trader EJB and print results
            Hello hello=tradeRef.create();
            String tr=hello.say();
            System.out.println("Result = "+tr);
        }
        catch (Exception e) {
            System.out.println("ERROR / "+e);
            e.printStackTrace(System.out);
        }
    }
}
```

3. Compilation

Compile the generated files.

WARNING: Compile the file corresponding to the client parts, the files Hello.java, HelloHome.java, _Exception.java, ..., and _*Stub.java, *Helper.java, *ValueFactory.java, *Operation.java (* represents the name of the interface).

Accessing a CORBA service by an EJB deployed on JOnAS server

CORBA Service

Create the CORBA service.

Create the idl file corresponding to this service (e.g. the interface name, which is "Hello").

Generate the java file corresponding to the idl service with the `idlj −fall Hello.idl` tool.

Implement the java interface (in this example, the service will be bound with the name "Hello" in the server implementation).

Start the orb.

Start the CORBA service.

EJB on JOnAS

To call the CORBA service, generate the java file corresponding to the idl file.

For this, apply the idlj tool on the idl file corresponding to the CORBA service description:.

```
idlj −fclient Hello.idl
```

Then, create an EJB.

For calling the CORBA service, initialize the orb by specifying the host and the port.

Then, get the environment.

Get the java object corresponding to the CORBA service with the environment.

Call the method on this object.

Example code:

```
try {
          String[] h=new String[4];
          h[0]="−ORBInitialPort";
          h[1]=port;
          h[2]="−ORBInitialHost";
          h[3]=host;

          ORB orb=ORB.init(h,null);

          // get a reference on the context handling all services
          org.omg.CORBA.Object objRef=orb.resolve_initial_references("NameService");

          NamingContextExt ncRef=NamingContextExtHelper.narrow(objRef);
          Hello hello=HelloHelper.narrow(ncRef.resolve_str("Hello"));
          System.out.println(hello.sayHello());
          return hello.sayHello();
          }
```

```
catch (Exception e) {
    ...
}
```

Howto: How to migrate the New World Cruises application to JOnAS

This guide describes the modifications required for migrating the J2EE application New World Cruise to JOnAS server.

The content of this guide is the following:

JOnAS configuration

The first step is to configure the database used for this application. Copy the file <db>.properties to the directory $JONAS_BASE/conf. Edit this file to complete the database connection.

Then, modify the JOnAS DBM Database service configurations in the file $JONAS_BASE/conf/jonas.properties, to specify the file containing the database connection.

New World Cruise Application

EJB modification code

To be EJB2.0−compliant, add the exceptions RemoveException and CreateException for EJB's methods ejbRemove and ejbCreate.

Additionally, the GlueBean class uses a local object in GlueBean constructor. However, it must use a remote object because it is a class calling an EJB. Therefore, modify the comment in this class with the following:

```
// If using the remote interface, the call would look like this
cruiseManagerHome = (CruiseManagerHome)
     javax.rmi.PortableRemoteObject.narrow(result, CruiseManagerHome.class);
// Using the local interface, the call looks like this
//cruiseManagerHome = (CruiseManagerHome) result;
```

EJB's Deployment descriptor

There are three EJBs, thus there must be three ejb−jar.xml files that correspond to the EJB's deployment descriptors and three jonas−ejb−jar.xml files that correspond to the JOnAS deployment descriptors.

First, rename the files <ejb_name>.ejbdd with <ejb_name>.xml; these files contain the EJB deployment descriptors.

Create the three jonas−<ejb_name>.xml files corresponding to the EJBs.

For the two entity Beans (Cruise and CruiseManager), describe the mapping between:

− the EJB name and jndi name (jndi name =ejb/<ejb name>),

− the jdbc and the table name,

− the EJB field and the table field, (the version of CMP is not specify in ejb−jar and JOnAS by default uses CMP1.1).

For the session Bean, describe the mapping between:

− the EJB name and jndi name (jndi name =ejb/<ejb name>)

Web Application

Create the jonas−web.xml that corresponds to the deployment descriptor of the New World Cruise application. Package the jonas−web.xml and the files under the directory Cruises/cruise_WebModule in the war file.

Build Application

Build the ear corresponding to the application.

This ear contains the three files corresponding to the three EJBs, as well as the web application.

SUN web service

Axis classes generation

To call a web service, first generate axis classes. The generated classes will allow a web service to be called using the static method.

For this step, download the file AirService.wsdl that corresponds to the SUN web service description or use the URL containing this file.

Then use the command:

```
java org.apache.axis.wsdl.WSDL2java <file_name>
```

This command generates four java files:

* AirService.java: the service interface.

* AirServiceLocator.java: the service implementation

* AirServiceServantInterface: the endpoint interface

* AirServiceServantInterfaceBindingStub.java: the stub class

To call the SUN web service, instantiate the service implementation. Then call the method getAirService() to get the end point, and call the appropriate method.

```
AirService airService=new AirServiceLocator();
AirServiceServantInterface interface=airService.getAirService();
Object result=interface.<method>;
```

JSP files

The file Part2_site.zip contains the web application that uses the SUN web service.

It includes several jsp files that must be modified to use the axis classes.

As an example, make the following replacements in the index.jsp file:

```
// Get our port interface
AirPack.AirClientGenClient.AirService service =
          new AirPack.AirClientGenClient.AirService_Impl();
AirPack.AirClientGenClient.AirServiceServantInterface port =
          service.getAirServiceServantInterfacePort();

// Get the stub and set it to save the HTTP log.
AirPack.AirClientGenClient.AirServiceServantInterface_Stub stub =
          (AirPack.AirClientGenClient.AirServiceServantInterface_Stub) port;
java.io.ByteArrayOutputStream httpLog =
             new java.io.ByteArrayOutputStream();
stub._setTransportFactory
   (new com.sun.xml.rpc.client.http.HttpClientTransportFactory(httpLog));

// Get the end point address and save it for the error page.
String endPointAddress = (String)
      stub._getProperty(stub.ENDPOINT_ADDRESS_PROPERTY);
request.setAttribute("ENDPOINT_ADDRESS_PROPERTY", endPointAddress);
```

by

```
// Get our port interface
AirService_pkg.AirService service = new AirService_pkg.AirServiceLocator();
AirService_pkg.AirServiceServantInterface port =
         service.getAirServiceServantInterfacePort();
```

Additionally, the Exception

```
throw new com.sun.xml.rpc.client.ClientTransportException(null, new Object[] {e});
```

is replaced by

```
 throw new Exception(e);.
```

Web Application

Finally, create the web application (jonas−web.xml) and reuse the web.xml that is in Part2_site.zip. Then, build the web application, which contains:

```
    META−INF/
    META−INF/MANIFEST.MF
    WEB−INF/
    WEB−INF/jonas−web.xml
    WEB−INF/lib/
    WEB−INF/lib/CruiseManager.jar
    WEB−INF/classes/
    WEB−INF/classes/AirService_pkg/
    WEB−INF/classes/AirService_pkg/AirServiceServantInterface.class
    WEB−INF/classes/AirService_pkg/AirServiceServantInterfaceBindingStub.class
    WEB−INF/classes/AirService_pkg/AirService.class
    WEB−INF/classes/AirService_pkg/AirServiceLocator.class
    PalmTree.jpg
    aboutus.jsp
    air_icon.gif
    airbook.jsp
    airclient.jsp
    airdates.jsp
    airdone.jsp
    airlist.jsp
    clear.gif
    crubook.jsp
    crudone.jsp
    cruise_icon.gif
    cruises.jsp
    flights.jsp
    index.jsp
    nwcl_banner.gif
    nwcl_banner_a.gif
    nwcl_styles.css
    WEB−INF/web.xml
```

JOnAS web service

Deployment

This web service uses the EJB stateless CruiseManager. To deploy this web service, create the web service deployment descriptor:

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

    <!--
        AXIS deployment file for EJB Cruise
    -->
     <service name="AirService" provider="java:EJB">

        <!--
            JNDI name specified in jonas-CruiseApp.xml
        -->
        <parameter name="beanJndiName"
                       value="ejb/CruiseManager"/>

        <!--
            use of remote interfaces to access the EJB is allowed
        -->
          <parameter name="homeInterfaceName"
                       value="cruisePack.CruiseManagerHome"/>
          <parameter name="remoteInterfaceName"
                       value="cruisePack.CruiseManager"/>


        <!--
            Specify here allowed methods for Web Service access (* for all)
        -->
        <parameter name="allowedMethods"
                 value="createPassenger,getAllDates,getByDepartdate"/>

        <typeMapping
            xmlns:ns="urn:AirService/types"
            qname="ns:ArrayOfString"
            type="java:java.lang.String[]"
            serializer="org.apache.axis.encoding.ser.ArraySerializerFactory"
            deserializer="org.apache.axis.encoding.ser.ArrayDeserializerFactory"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        />
    </service>
  </deployment>
```

To deploy this web service, first deploy the web application axis.war and the EJB corresponding to the web service (CruiseManager.jar).

Then, deploy the web service using the axis client:

```
    jclient org.apache.axis.client.AdminClient
               -lhttp://localhost:<port>/<context-root-axis.war>/servlet/AxisServlet <ws_wsdd>
```

Axis classes generation

To call a web service, first generate axis classes. The generated classes will allow a web service to be called using the static method.

For this step, download the file AirService.wsdl corresponding to the SUN web service description or use the URL containing this file.

The use of the command is as follows:

```
java org.apache.axis.wsdl.WSDL2java <file_name or url>
```

This command generates four java files:

* CruiseManagerService.java: the service interface

* CruiseManagerServiceLocator.java: the service implementation

* CruiseManager.java: the endpoint interface

* AirServiceSoapBindingStub.java: the stub class

To call the JOnAS web service, instantiate the service implementation. Then, call the method getAirService() to get the end point interface, and call the appropriate method.

```
AirService_JOnAS.Client.CruiseManagerService cms=
     new AirService_JOnAS.Client.CruiseManagerServiceLocator();

AirService_JOnAS.Client.CruiseManager cmi=cms.getAirService();

Object result=cmi.<method>;
```

JSP files

To access the JOnAS web service, copy the jsp files contained in the EJB's web application (Cruises/cruise_WebModule).

The JOnAS web service call must replace the call for each EJB.

Web Application

Finally, create the web application: the jonas−web.xml. Then, build the web application, which contains:

```
META−INF/
META−INF/MANIFEST.MF
WEB−INF/
```

```
WEB-INF/jonas-web.xml
WEB-INF/lib/
WEB-INF/lib/CruiseManager.jar
WEB-INF/classes/
WEB-INF/classes/AirService_pkg/
WEB-INF/classes/AirService_JOnAS/Client/CruiseManagerService.class
WEB-INF/classes/AirService_JOnAS/Client/AirServiceSoapBindingStub.class
WEB-INF/classes/AirService_JOnAS/Client/CruiseManager.class
WEB-INF/classes/AirService_JOnAS/Client/CruiseManagerServiceLocator/AirServiceLocator.class
PalmTree.jpg
aboutus.jsp
air_icon.gif
airbook.jsp
airclient.jsp
airdates.jsp
airdone.jsp
airlist.jsp
clear.gif
crubook.jsp
crudone.jsp
cruise_icon.gif
cruises.jsp
flights.jsp
index.jsp
nwcl_banner.gif
nwcl_banner_a.gif
nwcl_styles.css
WEB-INF/web.xml
```

Howto: Execute JOnAS as a WIN32 Service

This document describes the procedures necessary to run JOnAS as a system service on Microsoft Windows platforms. This applies starting from JOnAS 3.3.2.

Instructions

This procedure uses ANT targets that are introduced in JOnAS 4.0. The procedure also uses the Java Service Wrapper open source project which must be downloaded and installed separately.

Download and install Java Service Wrapper

1. Download Java Service Wrapper version 3.0.5 or later, and unzip the package to a directory in the local filesystem.
2. Set `WRAPPER_HOME` environment variable to the root directory for Java Service Wrapper.

   For example, if the package for Wrapper version 3.0.5 is unzipped into c:\jsw, then SET WRAPPER_HOME=c:\jsw\wrapper_win32_3.0.5

create_win32service

Before JOnAS can be run as a WIN32 service, it is necessary to copy Java Service Wrapper executable files to the %JONAS_BASE% directory and create a Java Service Wrapper configuration file. Prior to executing the steps in this section, it is necessary to create a JONAS_BASE directory as described in the JOnAS Configuration Guide.

1. Verify that JONAS_BASE and WRAPPER_HOME environment variables are set.
2. Set %JONAS_ROOT% as the current directory.
3. Execute **`ant [-Djonas.name=<server_name>] create_win32service`**.

   The `-Djonas.name=<server_name>` parameter is optional. If not specified, the default server name is 'jonas'.

   NOTE: it is necessary to execute `create_win32service` to regenerate wrapper configuration files whenever the JOnAS configuration is modified. Refer to the Modify JOnAS Configuration section for more information.

install_win32service

After the %JONAS_BASE% directory has been updated for use with Java Service Wrapper, JOnAS can be installed as a WIN32 service using the `install_win32service` ant target. Prior to installing the configuration as a WIN32 service, the configuration can be tested as a standard console application. Refer to the Testing configuration section for more information. The following steps will install the service.

1. Verify that JONAS_BASE and WRAPPER_HOME environment variables are set.
2. Set %JONAS_ROOT% as the current directory.
3. Execute **`ant install_win32service`**.

By default, the service is configured to start automatically each time Windows starts. If the administrator would prefer to start the service manually, modify the `wrapper.ntservice.starttype` parameter in the %JONAS_BASE%\conf\wrapper.conf file. Set the value as described in the comments found in the wrapper.conf file.

uninstall_win32service

When it is no longer desirable to run JOnAS as a Windows service, the service can be uninstalled using the `uninstall_win32service` ant target.

1. Verify that JONAS_BASE and WRAPPER_HOME environment variables are set.
2. Set %JONAS_ROOT% as the current directory.
3. Verify that the service has been stopped.
4. Execute **`ant uninstall_win32service`**.

start JOnAS service

To start the JOnAS service, open the Service Control Manager (Control Panel Services) window, select the JOnAS service and start the service.

By default, JOnAS will be started automatically each time Windows is started. After installing the service, it can be started manually to avoid the need to reboot Windows.

stop JOnAS service

To stop the JOnAS service, open the Service Control Manager window, select the JOnAS service and stop the service.

Files managed by create_win32service

The `create_win32service` ant target copies executable files from the Java Service Wrapper installation directory and generates a configuration file in the %JONAS_BASE% directory. The following files are managed by the `create_win32service` ant target.

- bin\Wrapper.exe
- bin\server.bat
- bin\InstallService−NT.bat
- bin\UninstallService−NT.bat

- lib\wrapper.jar
- lib\wrapper.dll

- conf\wrapper.conf
- conf\wrapper_ext.conf

NOTE: `wrapper_ext.conf` contains Java Service Wrapper configuration properties specific to the JOnAS service. This file is generated by the `create_win32service` ant target. Any changes made to this file will be lost when the `create_win32service` target is executed.

Modify JOnAS Configuration

Most of the JOnAS configuration is specified using property and XML files in the %JONAS_BASE%\conf directory. Changes to the files located in the %JONAS_BASE%\conf directory take effect the next time the service is started. It is only necessary to stop the service and restart the service for the changes to take effect.

In addition to the files located in the conf directory, JOnAS configuration is affected by the contents of %JONAS_ROOT%\bin\nt\config_env.bat, and by the CLASSPATH and JAVA_OPTS environment variables. If changes are made to config_env.bat, or to the CLASSPATH or JAVA_OPTS environment variables, it is necessary to update the Java Service Wrapper configuration files.

1. Using the Windows Service Control Manager, stop the JOnAS service.
2. Update the Java Service Wrapper configuration. Refer to the create_win32service section for details.
3. Test the updated configuration. Refer to the Testing configuration section for more information.
4. Using the Windows Service Control Manager, start the JOnAS service.

Testing configuration

After the Java Service Wrapper configuration files have been generated, it is possible to test the configuration in a console window before installing the configuration as a WIN32 service.

1. Verify that JONAS_BASE environment variable is set.
2. Execute **`%JONAS_BASE%\bin\server`**.

The Java Service Wrapper will start as a console application and load JOnAS using the configuration generated by the `create_win32service` ant target.

Enter **`CTRL-C`** to terminate JOnAS. After pressing Ctrl-C, the Java Service Wrapper displays the following messages to the execution report, and/or log file.

```
wrapper  | CTRL-C trapped.  Shutting down.
jvm 1    | 2003-12-02 15:25:20,578 : AbsJWebContainerServiceImpl.unRegisterWar : War /G:/w32svc/webapp
jvm 1    | Stopping service Tomcat-JOnAS.
wrapper  | JVM exited unexpectedly while stopping the application.
wrapper  | <-- Wrapper Stopped.
```