Seminario di Ingegneria del Software

Apache ServiceMix

Francesco D'Addio, Danilo Ricci

Indice

1. Introduzione

L’esplosione di internet, avvenuta negli anni novanta, è andata di pari passo con il forte interesse che le aziende hanno mostrato verso questo nuovo strumento. In particolare, dopo aver visto una forte diffusione dei sistemi informativi all’interno delle aziende, si è sentita la necessità di poter integrare piattaforme differenti.
Questo tipo di esigenza sorge non solo tra aziende, ma addirittura all’interno della medesima azienda nella quale, per motivi storici e organizzativi, diverse divisioni hanno operato scelte tecnologicamente differenti nella realizzazione del proprio sistema informativo.
Dal punto di vista organizzativo è plausibile immaginare che ogni azienda possa fornire servizi e al contempo utilizzarne. D’altro canto dal punto di vista tecnologico affinché questa interoperabilità sia fattibile, le aziende devono accordarsi su un linguaggio comune di descrizione dei servizi in modo tale da riconoscere cosa un sistema mette a disposizione. Ciò viene accompagnato anche da un meccanismo di ricerca dei servizi esistenti e dalla possibilità di utilizzare il Web come canale di trasmissione.
Una soluzione tecnologica adatta all’interoperabilità di sistemi è rappresentata dai Web Services (WS), il cui ruolo non si limita solo a tale discorso, bensì permette di descrivere nuovi servizi realizzati *ad hoc*, sempre però con l’intento di fornire una soluzione *platform-independent*. I WS nascono dall’evoluzione di un’architettura orientata ai servizi, SOA, nata per integrare i rapporti Business to Business. Attualmente, per i WS, esistono varie specifiche in grado di supportare scenari differenti.
Un’altra soluzione tecnologica è rappresentata dall’ **Enterprise Service Bus** (**ESB**); questa è un'infrastruttura software che fornisce servizi di supporto ad architetture SOA complesse. Un ESB si basa su sistemi disparati, interconnessi con tecnologie eterogenee, e fornisce in maniera consistente servizi di *orchestration*, sicurezza,

messaggistica, routing intelligente e trasformazioni, agendo come una dorsale attraverso la quale viaggiano servizi software e componenti applicativi. Un ESB si contraddistingue come soluzione migliorativa, rispetto ad altre più classiche di tipo *SOA oriented* in quanto ad esso sono delegati i servizi comuni denominati *core service* che andrebbero altresì realizzati.

In questa tesina abbiamo studiato gli Enterprise Service Bus in particolare Apache ServiceMix. ServiceMix costituisce un ESB open source, che, ad oggi, risulta essere il più maturo. La sua realizzazione si basa sulla specifica Java Business Integration (JBI), che costituisce un'architettura in grado di garantire integrazione tra i servizi.

Lo scopo di questa tesina è stato capire se era possibile usare ServiceMix come Web Service cercando di capire le interazioni tra questi due componenti.

Quindi abbiamo svolto due esempi:

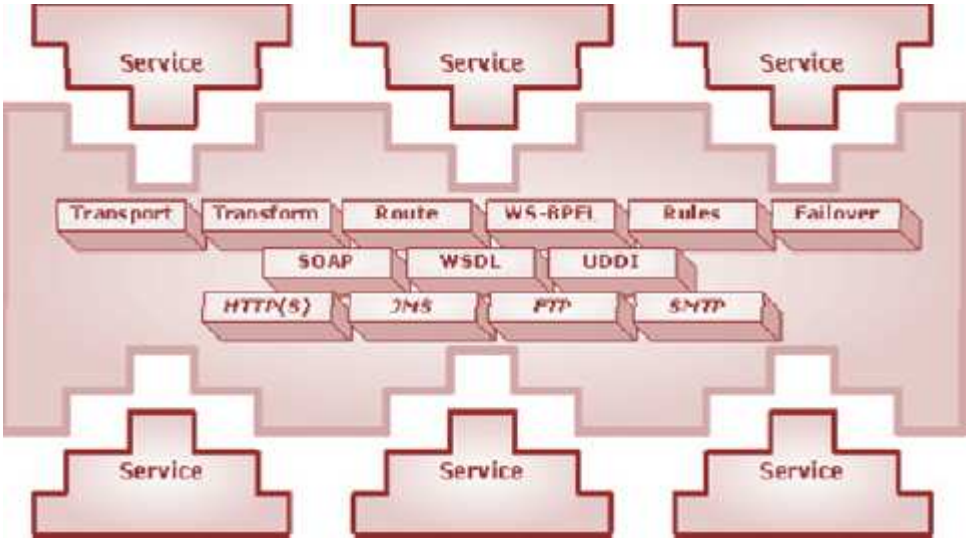
- il primo mostra come ServiceMix possa essere utilizzato come Web Service
- il secondo, ha l'intento di proporre un esempio di utilizzo standard di un ESB, nel caso particolare di realizzare un'applicazione che si interfaccia tramite JMS

2. Enterprise Service Bus e Java Business Integration

2.1. Enterprise Service Bus

Un Enterprise Service Bus (ESB), può esser definito come un middleware in grado di riunire varie tecnologie di integrazione, al fine di creare servizi di business ampiamente disponibili per il riuso. ESB offre la migliore soluzione per venire incontro alla *sfida* di integrazione di applicazioni aziendali odierne, fornendo un'infrastruttura software, che permette l'impiego di architetture SOA.
ESB fornisce un ambiente di esecuzione per il deploy e l'attivazione dei servizi, associati a *tools* di progetto in grado di definirne le interazioni. In particolare, i servizi non interagiscono direttamente l'un l'altro, bensì, ESB si comporta come un mediatore tra essi, garantendone il disaccoppiamento.
I servizi chiave forniti da un ESB includono:

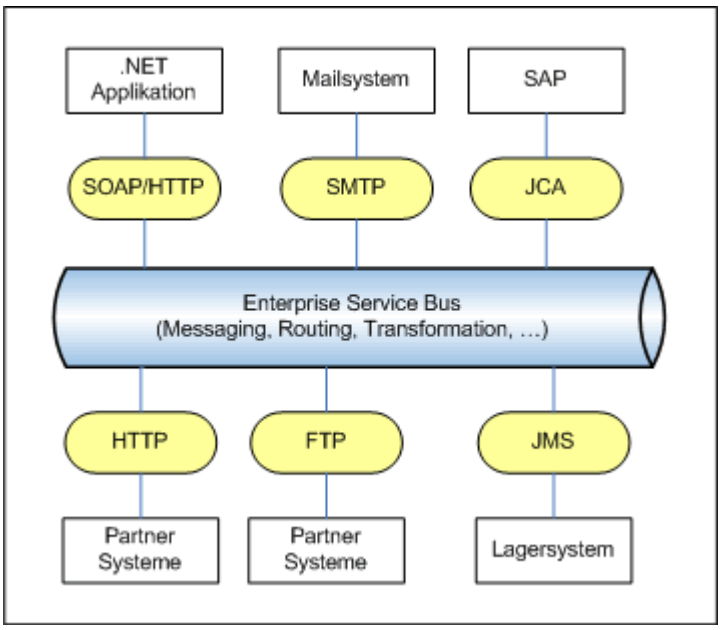
- protocolli di trasporto per i servizi
- definizione e scoperta dei servizi *deployati*
- gestione ed instradamento dei messaggi



Molti fornitori ESB basano la loro proposta SOA su tecnologie e open standards, inclusi i web services. Questi forniscono una varietà di protocolli di trasporto per l'invocazione dei servizi, tra cui HTTP, FTP e JMS. I fornitori ESB mettono a disposizione anche caratteristiche di *quality of service*, come *fault tolerance*, *fai lover, load balancing, message buffering* etc.

Enterprise Service Bus costituisce, dunque, un nuovo approccio per l'integrazione, in grado di fornire una rete altamente distribuita e debolmente accoppiata.

Nelle aziende che utilizzano architetture *event-driven*, gli eventi di business, che riguardano il normale corso di un processo commerciale, possono verificarsi in qualsiasi ordine e in un qualunque momento. Pertanto, le applicazioni che scambiano dati hanno bisogno di comunicare l'una con l'altra, utilizzando un'architettura *event-driven SOA*, così da avere la capacità di reagire ai cambiamenti delle necessità del business. In un ESB, le applicazioni e i servizi *event-driven* sono raggruppati in SOA in maniera disaccoppiata, ciò consente loro di operare indipendentemente.



ESB supporta:

- l'eterogeneità dei messaggi, intesa sia in termini di *molteplicità di modelli* (sincroni, asincroni, publish e subscribe), sia in termini di *molteplicità di formati* (e-mail, JMS, SOAP, XML).

- molteplici protocolli di trasporto per l’instradamento (File, FTP, HTTP, JMS, E-mail).
- scoperta dei servizi, memorizzando informazioni su di essi, i relativi schemi, WSDLs e politiche, garantendo una gestione centralizzata e un accesso distribuito.

Si fa spesso riferimento ai requisiti minimi di un ESB, come sistema di consegna di messaggi, tramite l’acronimo TRANS, che definisce un ESB come un’entità software in grado di:

- ***T***rasformare i messaggi da un formato ad un altro.
- ***R***, inoltrare i messaggi ai servizi registrati, garantendo quality-of-service.
- ***A***rricchire il contenuto del messaggio con informazioni riguardanti il richiedente.
- ***N***otificare i listeners registrati.
- ***S***, assicurare la consegna del messaggio.

2.2. Java Business Integration

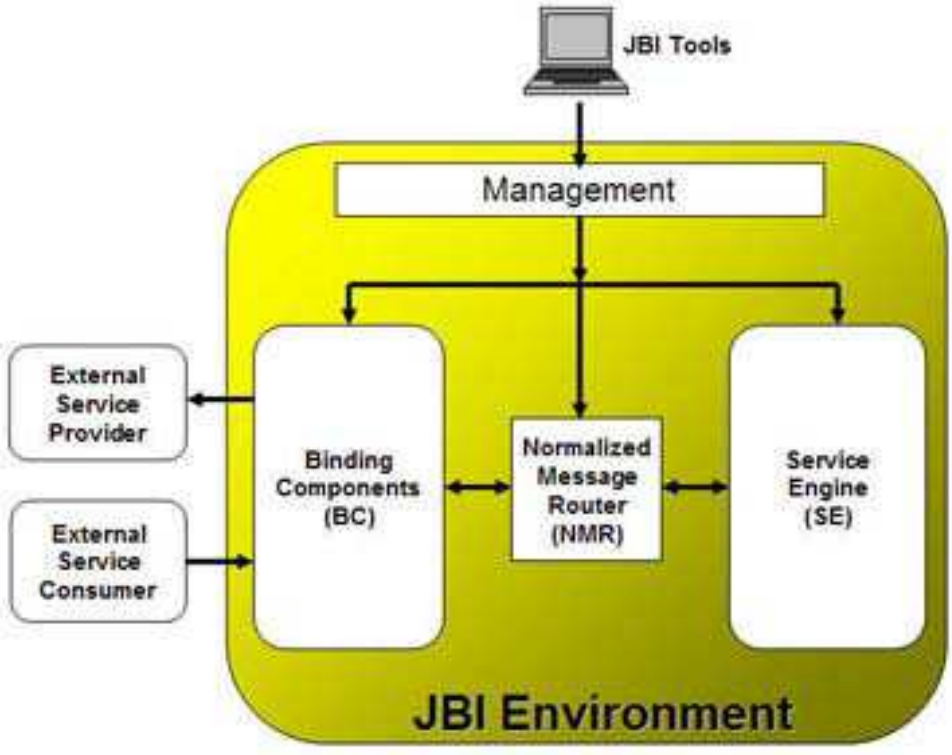
Java Business Integration (JBI) definisce un'architettura standard per uniformare l'integrazione di componenti eterogenei all'interno di applicazioni JBI-compliant. L'idea di base è di potere definire dei contenitori di servizi che consentano la System Integration e permettere quindi a sistemi inizialmente non progettati per lavorare insieme di cooperare tra loro come se fossero un'unica applicazione. In questo modo lo sviluppatore sarà in grado di sviluppare un'applicazione "assemblando" le funzionalità necessarie, utilizzando gli opportuni Componenti plug-in JBI. I Componenti plug-in JBI forniscono la logica di integrazione per connettere applicazioni Java con applicazioni non-Java, risorse legacy e pacchetti applicativi. JBI si pone quindi un obiettivo molto ambizioso: ottenere servizi d'integrazione

riusabili e portabili su qualsiasi prodotto JBI-compliant, ottenendo l'effetto Java "Write Once, run everywhere" nell'ambito dell'integrazione.

Per l'interazione tra i servizi, JBI fornisce interfacce *"ben definite"*. Tali interfacce sono esposte dai servizi, affinché JBI possa instradare i messaggi tra essi. Pertanto, JBI agisce come un mediatore tra i servizi.

Il *cuore esecutivo* dell'architettura JBI, comprende principalmente, all'interno della stessa Java Virtual Machine, i seguenti componenti:

- *Component Framework:* consente il deploy dei diversi tipi di componenti all'interno dell'ambiente JBI.
- *Normalized Message Router:* fornisce un meccanismo standard per lo scambio dei messaggi tra i servizi.
- *Management Framework:* basato su JMX, permette il deploy, la gestione e il monitoraggio dei componenti all'interno dell'ambiente JBI.



JBI definisce due tipi di componente :

- *Service Engine Components (SE):* costituiscono i componenti responsabili dell'implementazione della business logic. Tali componenti possono essere realizzati internamente utilizzando varie tecnologie e principi di progetto.

- *Binding Components (BC):* costituiscono i componenti principalmente utilizzati per fornire collegamento a livello trasporto, tra i servizi deployati.

L'aspetto chiave delle architetture JBI è il disaccoppiamento dei servizi, cosicché, la business logic non risulta essere caricata dai dettagli dell'infrastruttura, richiesti per l'invocazione ed il consumo dei servizi. Ciò garantisce un'architettura flessibile ed estensibile.

All'interno di JBI, sia i Binding Components che i Service Engine Components, possono assumere il ruolo di fornitori di servizi e/o consumatori di servizi.

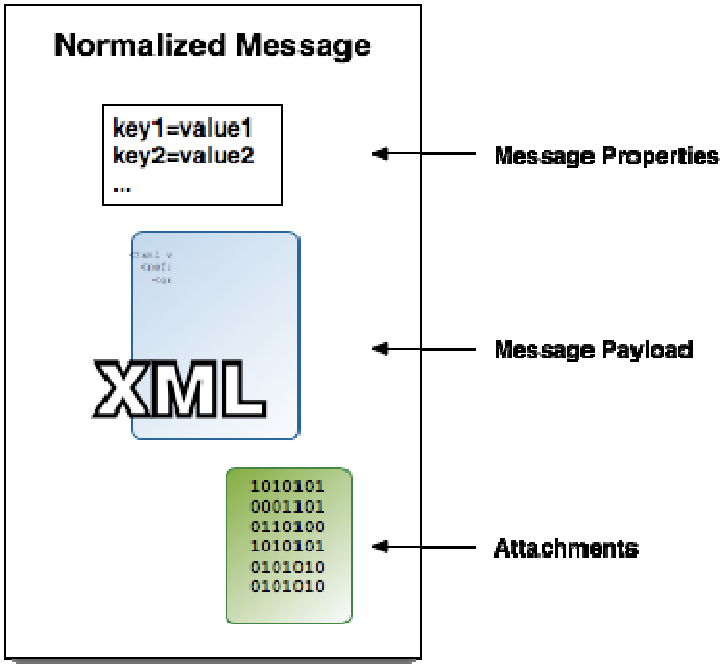
JBI definisce un modello di scambio messaggi che disaccoppia i consumers dai providers. Tale modello è definito utilizzando WSDL2. WSDL viene usato per descrivere le operazioni esposte sia dai componenti SE che dai componenti BC.

Uno dei componenti chiave utilizzati nell'architettura JBI è il *Normalized Message Router* (NMR). Il NMR effettua il routing dei messaggi dai BC verso i SE nelle comunicazioni *inbound* e dai SE verso i BC nelle comunicazioni *outbound*. I servizi presentano delle interfacce, che costituiscono un insieme di operazioni. Ciascuna operazione è composta da zero o più messaggi.

JBI utilizza un formato normalizzato per la rappresentazione dei messaggi all'interno dell'ambiente di sviluppo.

Un messaggio normalizzato è costituito dalle seguenti parti:

- *Proprietà del messaggio*: costituiscono i dati extra associati al messaggio. Possono contenere informazioni sulla sicurezza, informazioni su specifici componenti, etc.
- *Carico del messaggio*: racchiuso in un documento XML.
- *Allegati del messaggio*.

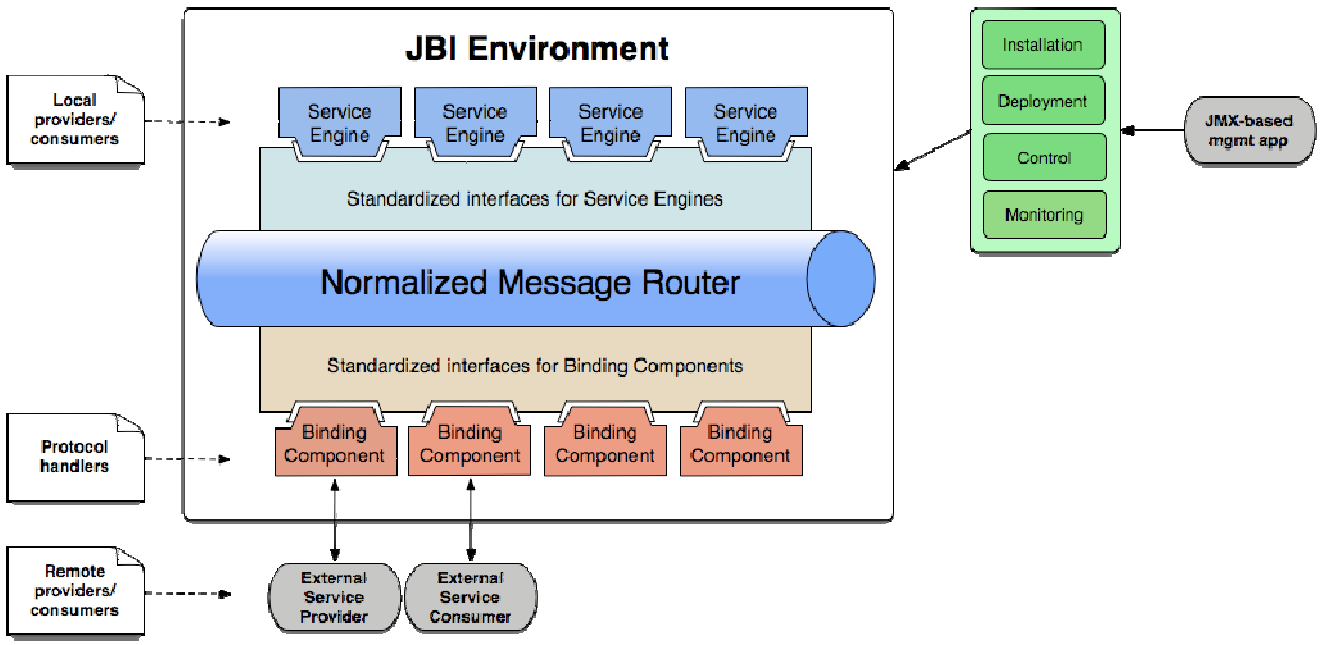


La specifica JBI fornisce interfacce standard ai consumers e ai providers per lo scambio di messaggi tramite NMR. NMR supporta sia una semantica di tipo *one-way* che di tipo *request-response* per l'invocazione di servizi.

Fondamento dell'interoperabilità dei componenti JBI è il pattern Message Exchange (MEP) . Tale pattern è definito in termini di tipo di messaggio (message type), che può essere *normal* o *fault*, e di direzione del messaggio. Ad ogni tipologia di invocazione (One Way, Request-Response, ...) è associato un ben preciso MessageExchange Pattern (In-Only, In-Out, ..).

I messaggi scambiati sono inoltrati attraverso un *delivery channel*, che rappresenta un canale di comunicazione bidirezionale, usato dai componenti BC e SE per trasportare i messaggi attraverso NMR. L'interfaccia "javax.jbi.messaging.DeliveryChannel" costituisce una sorta di *contratto* tra consumers, providers e NMR. Un consumer utilizza il proprio delivery channel per invocare il servizio, mentre un provider utilizza il proprio delivery channel per accogliere l'invocazione al servizio offerto. Un componente che funziona sia come consumer che come provider utilizzerà lo stesso delivery channel per entrambi i ruoli. Pertanto, le implementazioni del delivery channel dovranno supportare l'uso concorrente, di una data istanza, da parte di più threads.

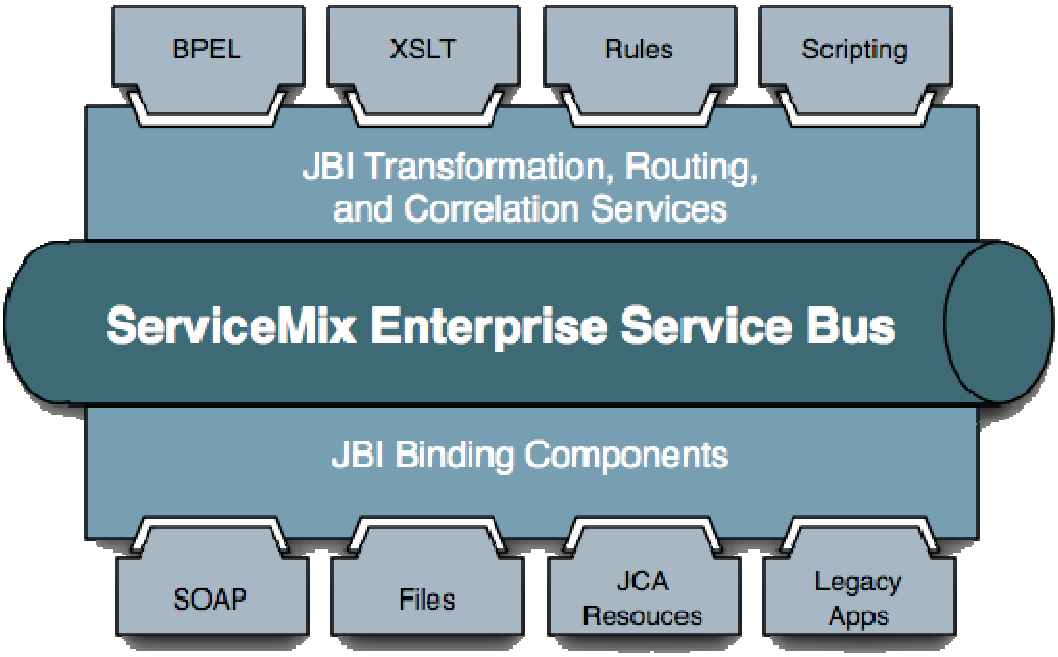
Scopo principale di JBI è, dunque, quello di inoltrare gli scambi tra un componente ed un altro. Binding Components e Service Engine interagiscono con NMR tramite il delivery channel, che fornisce un meccanismo di inoltro, per i messaggi, bidirezionale. Un service consumer esterno, inoltra una richiesta di servizio, attraverso uno specifico protocollo, ad un BC. Il Binding Component converte la richiesta in un messaggio normalizzato, quindi crea un pacchetto chiamato *Message Exchange* (ME) e lo invia attraverso il proprio delivery channel all'NMR, per inoltrarlo al service provider. Dopo la ricezione di un messaggio, il Binding Component crea un messaggio normalizzato, inserendolo all'interno di una nuova istanza MessageExchange e lo invia all'istanza di ServiceEndpoint. Dopo aver accettato il ME, il componente ServiceEndpoint, de-normalizza il messaggio e lo inoltra al service provider esterno.



3. Apache ServiceMix

3.1. Caratteristiche generali

ServiceMix è un ESB open source, distribuito, la cui realizzazione si basa sulle semantiche e le APIs (Application Programming Interface) della specifica Java Business Integration JSR-208.



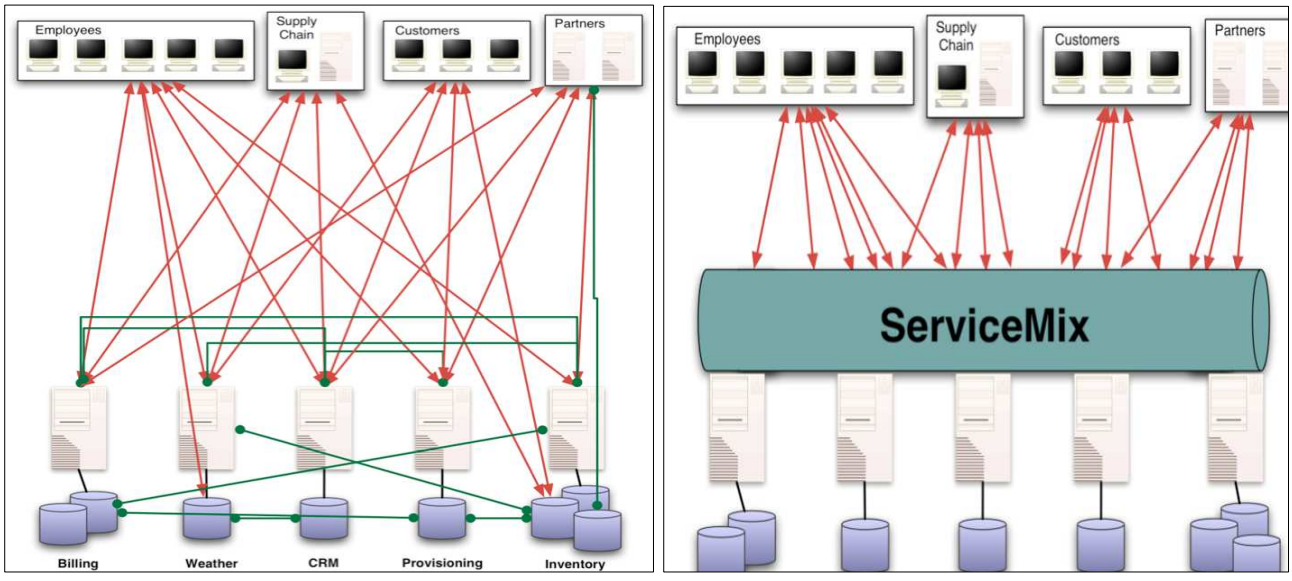
Open source e open standards-based tengono conto di un basso costo iniziale, della massima flessibilità, riuso e protezione degli investimenti. Ciò consente agli sviluppatori di utilizzare abilmente il lavoro di altri sviluppatori quando implementano componenti da inserire nell'ESB. Un gran numero di commercianti di terze parti possono fornire componenti e collegamenti conformi alla specifica open standard JBI. Questi componenti non solo interoperano l'uno con l'altro o con applicazioni esterne, tramite l'ESB di ServiceMix, ma possono semplicemente essere rimpiazzati con altri componenti, che forniscono servizi uguali o migliori. Gli

sviluppatori oltre a poter cambiare i componenti ESB, possono anche cambiare l'intero ESB e continuare ad utilizzare i componenti standard.

ServiceMix è leggero, facilmente integrabile ed utilizzabile in applicazioni Java SE o Java EE.

ServiceMix si relaziona a vari progetti open-source, in particolare l'architettura:

- è completamente integrata all'interno di Apache Geronimo, che consente agli sviluppatori di deployare servizi e componenti JBI direttamente su Geronimo. ServiceMix è stato certificato da JBI, come parte del progetto Geronimo.
- su applicazioni J2EE, è stata integrata all'interno di JBoss, per garantire più flessibilità.
- utilizza ActiveMQ per fornire servizi di *messaging* e *routing*.
- ha integrato il supporto Spring e può essere eseguito, all'interno di un client o di un server, come singolo ESB fornitore o come servizio all'interno di un altro ESB.



ServiceMix consente ai servizi di operare in maniera *event driven*. I servizi sono disaccoppiati e si mettono in ascolto, sul bus, delle richieste di servizio. Il bus è responsabile delle caratteristiche di Quality of Service (QoS), quali persistenza del messaggio, garanzia di consegna, gestione dei fallimenti, etc. Il progetto combina, efficacemente, le caratteristiche di SOA ed EDA per lo sviluppo ed il deploy di

applicazioni aziendali integrate. In particolare, supporta l'architettura guidata dagli eventi, per gli eventi che si verificano sia all'interno, che all'esterno del bus.
Internamente, ServiceMix distribuisce gli eventi utilizzando un'architettura di instradamento dei messaggi, chiamata *Flow*. In particolare, sono supportati, di default, tre tipi di Flow per l'instradamento dei messaggi:

- **STP**: interazioni di tipo *straight-through*, i componenti interagiscono direttamente tra loro.
- **SEDA**: (Staged Event Driven Architecture) per l'instradamento di messaggi scalabili.
- **CLUSTERED**: i componenti si registrano ad altre istanze ServiceMix, all'interno di un *cluster*, consentendo la propagazione di eventi distribuiti.

ServiceMix incapsula al suo interno un'ampia gamma di componenti riusabili, tra cui:

- WSIF per lavorare con una qualunque implementazione di "*Web Service Invocation Framework*".
- GROOVY per garantire un'integrazione agile e forte.
- HTTP, JMS per fornire un bus generale per l'instradamento dei messaggi.
- QUARTZ per l'integrazione di timer.

Per rendere semplice l'utilizzo di ServiceMix, da parte degli sviluppatori, è stata creata una JBI Client API, che facilita il lavoro con un qualunque JBI Container o con un qualunque componente JBI disponibile.

3.2. Componenti

ServiceMix viene fornito con una serie di componenti standard; questa è una breve descrizione di ognuno di essi:

- ***Servicemix-bean***

  Permette l'integrazione tra i beans (POJO) e il JBI bus per rendere più facile l'uso dei POJO processando i messaggi JBI scambiati sul bus.

  Come per un Message Driven Bean in J2EE, un POJO riceverà un messaggio dal NMR e lo processerà come vuole. A differenza di un componente JMS, dove il codice è già pronto, il componente Bean permetterà allo sviluppatore di creare qualsiasi tipo di messaggio che, però, dovrà essere creato di volta in volta.

- ***Servicemix-camel***

  Fornisce il supporto per l'utilizzo di Apache Camel, che fornisce una serie completa di Enterprise Integration Pattern per un routing flessibile e per la trasformazione in entrambi i codici Java o Spring XML per l'instradamento dei servizi sul Normalized Message Router.

- ***Servicemix-cxf-bc***

  ServiceMix si interfaccia con un Binding Component JBI compatibile con HTTP / SOAP o JMS / SOAP chiamato servicemix-cxf-bc che utilizza Apache cxf internamente.

- ***Servicemix-cxf-se***

  È un JBI Service Engine che espone (determinati) POJO come servizi sul bus JBI; utilizza Apache CXF internamente per eseguire invocazioni a servizi e xml marshalling.

- ***Servicemix-drools***

  Consente l'integrazione tra il JBI e il Drools Rules Engine. Questo Service Engine può essere usato per sviluppare un insieme di regole che implementeranno un router o un servizio vero e proprio.

  Come router, agirà soprattutto come proxy trasparente tra il consumer e il provider del servizio obiettivo.

  Come servizio, agirà come consumer scambiando messaggi con il client utilizzando i metodi forniti dal JBI helper.

- ***Servicemix-eip***

  È un contenitore di routing in cui diversi modelli di routing possono essere impiegati come Service Unit. Questo componente si basa sul libro “Enterprise Integration Pattern”.

- ***Servicemix-file***

  Prevede l'integrazione JBI con il file system. Può essere utilizzato per leggere e scrivere file tramite URI o per controllare periodicamente le directory per i nuovi file.

- ***Servicemix-ftp***

  Prevede l'integrazione JBI al server FTP; può essere utilizzato per leggere e scrivere file tramite FTP o per sondare periodicamente le directory per i nuovi file.

- ***Servicemix-http***

  ServiceMix si interfaccia con un Binding Component JBI compatibile con HTTP / SOAP chiamato servicemix-http.

- ***Servicemix-jms***

  ServiceMix si interfaccia con un Binding Component JBI compatibile con JMS chiamato servicemix-JMS

- ***Servicemix-jsr181***

  È un JBI Service Engine che espone (determinati) POJO come servizi sul bus JBI. Utilizza internamente Xfire per eseguire le invocazioni al servizio e per l’xml marshalling.

- ***Servicemix-lwcontainer***

  È un JBI Service Engine che accetta i file di configurazione servicemix.xml contenenti componenti leggeri.

- ***Servicemix-mail***

  Fornisce il supporto per la ricezione e l'invio di mail tramite il Enterprise Service Bus.

- ***Servicemix-osworkflow***

  Fornisce delle funzionalità per il controllo del flusso di lavoro per l'ESB.

  È possibile specificare uno o più flussi di lavoro e il loro processo avrà inizio quando sarà ricevuto un messaggio valido.

- ***Servicemix-quartz***

  È uno standard JBI Service Engine in grado di programmare e attivare i progetti utilizzando il Quartz scheduler.

- ***Servicemix-saxon***

  È uno standard JBI Service Engine di XSLT / XQuery. Si basa su Saxon e supporta XSLT 2.0 e XPath 2.0, e XQuery

- ***Servicemix-script***

  Prevede l'integrazione JBI con motori di scripting. Sfrutta il supporto Spring per i linguaggi dinamici.

- ***Servicemix-scripting***

  Fornisce il supporto per la trasformazione di script utilizzando JSR-223.

- ***Servicemix-wsn2005***

  È uno standard JBI Service Engine che attua la specifica WS-Notification tramite Oasis.

- ***Servicemix-validation***

  La componente di validazione ServiceMix schema prevede la convalida di documenti utilizzando JAXP 1.3 e XMLSchema o RelaxNG.

- ***Servicemix-xmpp***

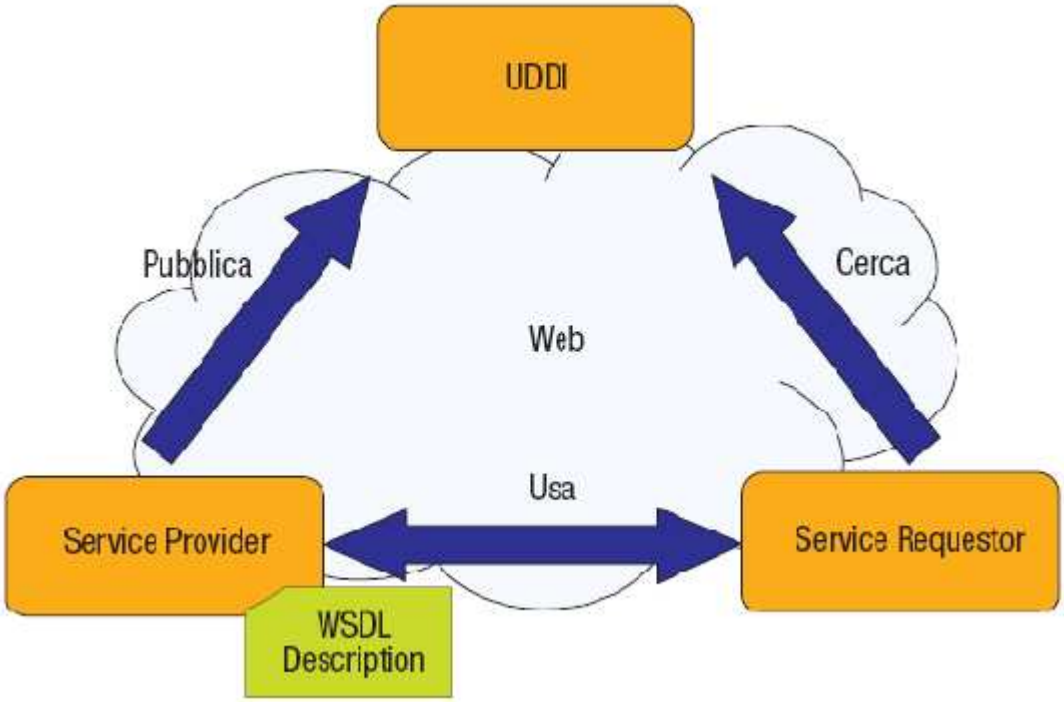
  Viene utilizzato per comunicare con XMPP (Jabber) server attraverso il JBI bus.

4. WebService e Java Message Service

4.1. WebService

I Web Services costituiscono una soluzione tecnologica adatta all'interoperabilità dei sistemi. Il ruolo dei Web Services non si limita solo ad un discorso di interoperabilità, bensì permette di descrivere nuovi servizi *ad hoc*, sempre, però, con l'intento di fornire una soluzione *platform-independent.*

Un'architettura per Web Services è un'istanza di una System Oriented Architecture, dove il mezzo di comunicazione considerato è il web. Le tre tecnologie fondamentali per Web Services sono: SOAP, WSDL e UDDI mentre il protocollo più comunemente utilizzato per creare collegamenti robusti è XML.



XML è attualmente considerate il linguaggio del futuro per lo scambio di dati. Si occupa della descrizione dei dati scambiati tramite Web Services. In realtà XML è un linguaggio a tag che permette di identificare il tipo di dato trasmesso associandone un significato.

SOAP costituisce il protocollo di accesso ai Web Services, basato su XML e HTTP, in grado di far interagire componenti remoti attraverso il web. La specifica delle chiamate viene descritta in XML, mentre HTTP è il protocollo di trasporto su cui poggia.

WSDL stabilisce un formato comune per descrivere e pubblicare informazioni Web Services. Gli elementi di WSDL descrivono dati, tramite XML, e le operazioni effettuate su di essi. Entrambi le parti partecipanti ad un'interazione devono avere accesso allo stesso WSDL per capirsi. Ovvero sia il fornitore che l'utente del servizio devono poter accedere allo stesso schema XML. La caratteristica principale di WSDL è quella di fornire un formato comune per codificare e decodificare messaggi a e da una qualunque applicazione virtuale.

UDDI, utilizzando opportune strutture dati, tiene traccia della dislocazione dei servizi e delle loro descrizioni. UDDI prevede la creazione di un ambiente distribuito peer-to-peer in cui i vari nodi, che contengono una parte dei servizi disponibili, possano interoperare tra loro, allo scopo di soddisfare le richieste di pubblicazione e ricerca dei servizi.

La ragione principale per la creazione e l'utilizzo di Web services è il "disaccoppiamento" che l'interfaccia standard, esposta dal Web Service, rende possibile fra il sistema utente ed il Web Service stesso. Per cui, modifiche di applicazioni possono essere attuate in maniera "trasparente". Tale flessibilità consente la creazione di sistemi software complessi, costituiti da componenti svincolati l'uno dall'altro e consente una forte riusabilità di codice e di applicazioni già sviluppate.

In particolare, i vantaggi principali dei Web Services sono i seguenti:

- permettono l'interoperabilità tra diverse applicazioni software, su diverse piattaforme hardware.
- utilizzano standard e protocolli "open"; i protocolli ed il formato dei dati è, dove possibile, in formato testuale, ciò li rende di più facile comprensione ed utilizzo da parte degli sviluppatori.

- mediante l'uso di HTTP, per il trasporto dei messaggi, i Web services non necessitano, normalmente, che vengano effettuate modifiche alle regole di sicurezza come filtro sui firewall.
- possono essere facilmente utilizzati in combinazione l'uno con l'altro (indipendentemente da chi li fornisce e da dove vengono resi disponibili) per formare servizi "integrati" e complessi
- consentono il riutilizzo di infrastrutture e di applicazioni già sviluppate.
- sono indipendenti dai sistemi operativi e dai linguaggi di programmazione.

4.2. Java Message Service

Java Message Service **(**JMS**)** è l'insieme di API che consentono lo scambio di messaggi tra applicazioni Java distribuite sulla rete. La prima specifica JMS è stata rilasciata nel 1998, mentre l'ultima versione delle API è la 1.1 pubblicata nel 2002.
In ambito JMS, un messaggio è un raggruppamento di dati che viene inviato da un sistema ad un altro. I dati sono solitamente utilizzati per notificare eventi e sono pensati per essere utilizzati da un programma su una macchina e non direttamente da un utente umano; in uno scenario distribuito un messaging service può essere pensato come un sistema distribuito di notifica di eventi.
Con il termine *messaging* ci si riferisce ad un meccanismo che consente la comunicazione asincrona tra client remoti:

- Un client invia un messaggio ad un ricevente (o ad un gruppo di riceventi)
- Il destinatario riceve il messaggio ed esegue le operazioni corrispondenti, in un secondo momento

In questo senso il sistema di messaggistica differisce da RMI in cui il mittente di un messaggio (per esempio il client che invoca un metodo remoto) deve attendere la risposta dal server che espone il metodo prima di poter continuare l'esecuzione.

JMS è basato sul paradigma peer-to-peer: un client può ricevere e spedire messaggi a qualsiasi altro client attraverso un provider: ciascun client si connette all'agente (gestore) della messaggistica che fornisce gli strumenti per creare, spedire, ricevere e leggere messaggi. In questo senso si può definire come messaging system ogni sistema che consente la trasmissione di pacchetti TCP/IP tra programmi client.
JMS permette una comunicazione distribuita del tipo *loosely coupled* (debolmente accoppiata): il mittente ed il ricevente, per comunicare, non devono essere disponibili allo stesso tempo. Grazie all'intermediazione del *messaging agent* (o server) i client che inviano/ricevono messaggi non hanno bisogno di avere conoscenza reciproca delle caratteristiche dell'altro per poter comunicare.
Differisce dalla comunicazione *tightly coupled* (per esempio quella usata in RMI) che, invece, richiede ad un'applicazione client di conoscere i metodi esposti su di un'interfaccia remota da un applicazione server; tutto ciò che il mittente ed il ricevente devono conoscere è il formato (*messageformat*) e la destinazione (*destination*) del messaggio.
JMS permette comunicazioni di tipo:

- ***Asincrono***

  Il JMS provider consegna i messaggi al client appena quest'ultimo si sia reso disponibile; il provider li consegnerà senza che il receiver abbia effettuato una richiesta specifica.

- ***Affidabile***

  JMS può assicurare che un messaggio sia consegnato una ed una sola volta.

- ***Sicuro***

  Tramite il cosiddetto *GuaranteedMessageDelivery* è possibile prevenire una perdita di informazioni in caso di malfunzionamento o di crash del message server, rendendo i messaggi persistenti prima di essere recapitati ai consumatori (per esempio mediante JDBC).

JMS non include, invece, sistemi di *Load balancing/Fault Tolerance* (la API non specifica un modo in cui diversi client possano cooperare al fine di implementare un

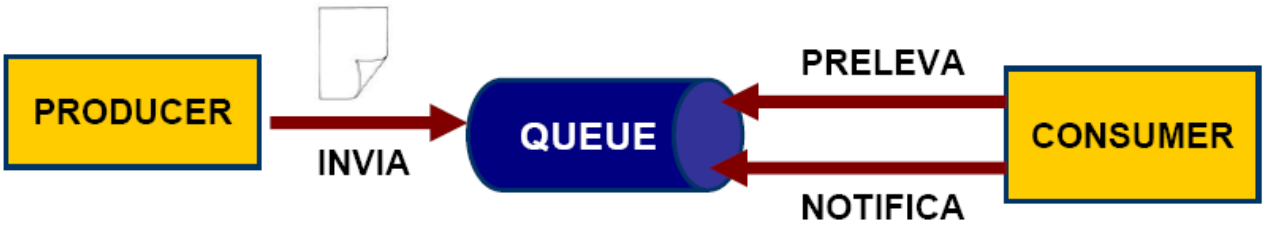
unico servizio critico), *notifica degli errori*, *amministrazione* (non definisce dei metodi per la gestione di prodotti di messaggistica) e *security* (non prevede API per il controllo della privacy e dell'integrità del messaggio).

Gli elementi chiave in una interazione JMS sono:

- *Client:* il programma che manda o riceve i messaggi JMS.
- *Messaggio:* ogni applicazione definisce un insieme di messaggi da utilizzare nello scambio di informazioni tra due o più client.
- *JMS provider:* sistema di messaggistica che implementa JMS e realizza delle funzionalità aggiuntive per l'amministrazione e il controllo della comunicazione attraverso messaggi.
- *Administered objects:* sono oggetti JMS preconfigurati, creati da un amministratore ad uso dei client. Incapsulano la logica specifica del JMS provider nascondendola ai client, garantendo maggiore portabilità al sistema complessivo.

JMS offre due modelli di messaging: *point-to-point* e *publish/subscribe*.

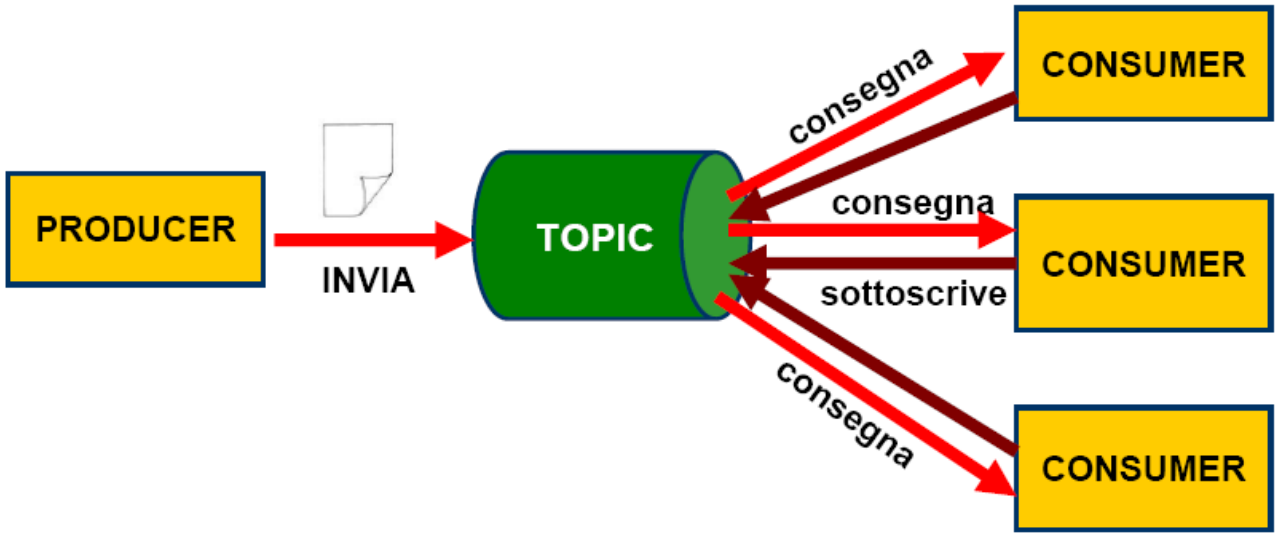
Il modello di messaggistica point-to-point (PTP o p2p) si basa sul concetto di coda: mittente (sender o producer) e destinatario (receiver o consumer). Ogni client JMS spedisce e riceve messaggi sia in modalità sincrona che in modalità asincrona, mediante canali virtuali conosciuti come "queues" (code). È un modello di tipo "pull-based", ovvero i messaggi vengono richiesti (prelevati) dalle code anziché essere consegnati ai client in maniera automatica.



Il modello PTP ha le seguenti caratteristiche: più produttori e più consumatori possono condividere la stessa coda, ma ogni messaggio ha solamente un "consumer", ovvero ogni messaggio può essere letto da un solo client; mittente e destinatario non

hanno nessuna dipendenza temporale, il ricevente notifica l'avvenuta ricezione e processamento del messaggio (acknowledge).

Nel modello publish/subscribe il producer può spedire un messaggio a molti consumers (One to Many), attraverso un canale virtuale chiamato *topic*.



Uno stesso topic può essere condiviso da più consumer ed utilizzato da più publisher; per ricevere i messaggi, i consumer devono "sottoscriversi" ad un topic, quindi qualsiasi messaggio spedito al topic viene consegnato a tutti i consumer sottoscritti, ciascuno dei quali riceve una copia identica di ciascun messaggio inviato al topic. È principalmente un modello "push-based": i messaggi vengono automaticamente inviati in broadcast ai consumer, senza che questi ne abbiano fatto esplicita richiesta.

Un messaggio JMS può essere consumato secondo due modalità:

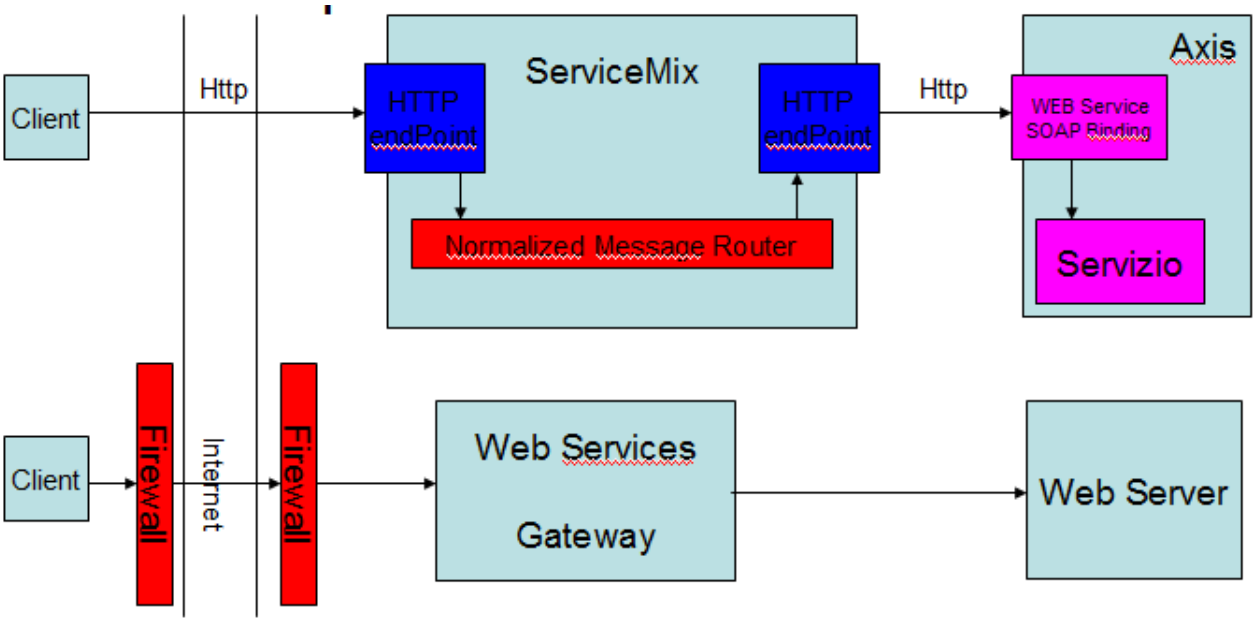
- *Modalità sincrona:* il subscriber (o il receiver) prelevano direttamente il messaggio dalla coda (operazione di fetch del messaggio), tramite l'invocazione del metodo *receive***().** Il metodo è sospensivo, il client rimane bloccato finché non arriva il messaggio o fino allo scadere di un timeout.
- *Modalità asincrona:* il client si registra presso un *messagelistener* attraverso un oggetto consumer. Se un messaggio arriva alla destinazione, il JMS provider consegna il messaggio attraverso il metodo *onMessage***()** del listener.

5. Costruire un WebService con Apache ServiceMix

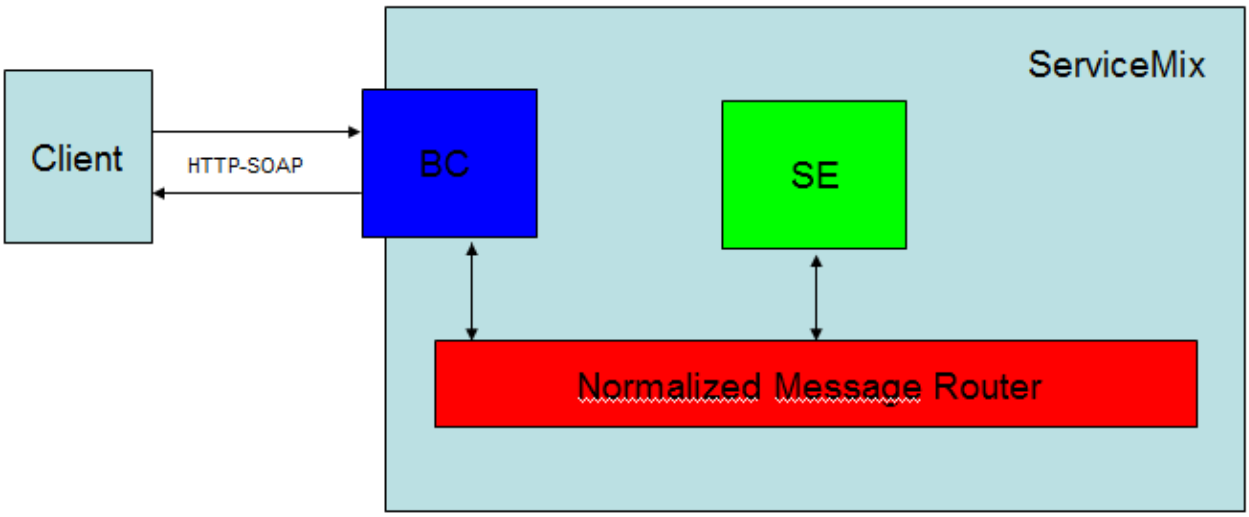
5.1. Specifica

ServiceMix può avere 2 diversi tipi di rapporto con Web Service:

- come semplice Gateway:



- come Web Service vero e proprio:



Abbiamo scelto di realizzare un esempio pratico su come trasformare ServiceMix in un Web Service; infatti utilizzare ServiceMix come semplice gateway non sfrutta a pieno tutte le potenzialità della piattaforma.

Per sviluppare questo progetto basta creare un BC compatibile con il protocollo http-soap che serve per fare interfacciare il Web Service con il mondo esterno. Quindi, una volta che arriveranno dal mondo esterno delle invocazioni a servizio, queste verranno prima normalizzate dal BC e poi attraverso l' NMR saranno inviate al servizio appropriato.

Il servizio sarà costituito da uno o più SE che processeranno la chiamata e la spediranno indietro sul NMR una volta finita la computazione.

I componenti di ServiceMix che utilizzeremo per il nostro esempio sono:

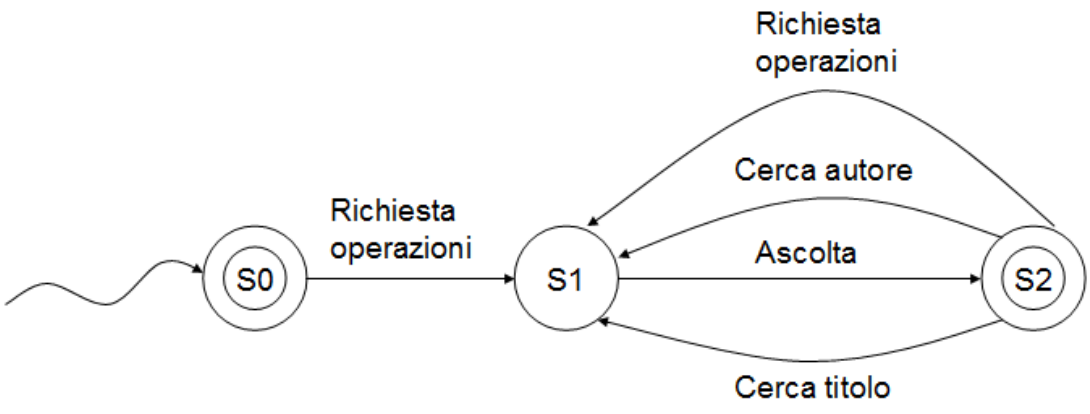
- Servicemix-cxf-bc
- Servicemix-cxf-se

Il client invierà al Web Service una richiesta per scoprire quali operazioni possono essere effettuate;

il Web Service offrirà al client la possibilità di effettuare le ricerche dei libri in due modalità, attraverso:

- L'autore
- Il titolo

Quindi il client sceglierà uno dei 2 metodi messi a sua disposizione e invierà la richiesta; infine il Web Service si connetterà al database (nel nostro caso un file txt) e restituirà al client il risultato.



5.2. Creazione del progetto

Ci sarà una cartella del progetto che conterrà tutte le sottocartelle per ogni SU – SA; quindi posizioniamoci in una cartella a scelta ed eseguiamo il seguente comando:

```
mvn archetype:create
-DarchetypeGroupId=org.apache.servicemix.tooling
-DarchetypeArtifactId=servicemix-project-root
-DgroupId=progetto.tesina
-DartifactId=wsdl-cxf-service
```

Questo comando genererà la cartella project-root chiamata “wsdl-cxf-service” contenente il pom.xml del progetto.

Ora spostandoci dentro la cartella appena creata invochiamo il comando:

```
mvn install
```

5.3. Creazione e configurazione del BC

ServiceMix mette a disposizione diversi Maven archetypes per aiutare a creare il progetto in modo più rapido e sicuro; utilizzeremo l’archetipo servicemix-cxf-bc-service-unit per creare la SU servicemix-cxf-bc del progetto.

Quindi dalla nostra cartella del progetto eseguiamo il seguente comando:

```
mvn archetype:create
-DarchetypeGroupId=org.apache.servicemix.tooling
-DarchetypeArtifactId=servicemix-cxf-bc-service-unit
-DgroupId=progetto.tesina
-DartifactId=cxf-bc-su
```

È stata creata la directory “cxf-bc-su” contenente:

- un file pom.xml file per la compilazione della SU del progetto
- una directory src/main/resources con all’interno:
  - un file xbean.xml per la configurazione della SU
  - un file service.wsdl per la descrizione del nostro servizio

Inoltre Maven nel file pom.xml nella cartella principale del progetto ha aggiunto il modulo:

```
<module>cxf-bc-su</module>
```

Il file pom.xml viene generato in automatico, occorre solamente modificare (per semplificare la leggibilità) il nome in:

```
<name> CXF BC SU del progetto</name>
```

I file service.wsdl e xbean.xml, invece, vanno modificati interamente; per semplicità basta sostituire questi (service.wsdl e xbean.xml) al posto di quelli generati automaticamente.

Di seguito discuteremo delle parti più significative di ognuno; iniziamo dal service.wsdl.

```
199  <wsdl:service name="LibriService">
200      <wsdl:port binding="tns:LibriSOAPBinding" name="soap">
201          <soap:address location="http://localhost:8080/LibriService/" />
202      </wsdl:port>
203  </wsdl:service>
204  </wsdl:definitions>
```

Nella riga 199, si definisce il servizio e, nella riga 201 si imposta l'indirizzo su cui il servizio sarà in attesa.

```
148  <wsdl:binding name="LibriSOAPBinding" type="tns:Libri">
149      <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
150      <wsdl:operation name="CercaAutore">
151          <wsdl:input>
152              <soap:body use="literal" />
153          </wsdl:input>
154          <wsdl:output>
155              <soap:body use="literal" />
156          </wsdl:output>
157          <wsdl:fault name="UnknownWord">
158              <soap:fault use="literal" name="UnknownWord" />
159          </wsdl:fault>
160      </wsdl:operation>
```

In quest'altro estratto, si evidenziano le definizioni di input, output e fault per l'operazione CercaAutore; le operazione CercaTitolo e RichestaOperazioni saranno del tutto simili.

```
122      <wsdl:portType name="Libri">
123          <wsdl:operation name="CercaAutore">
124              <wsdl:input message="tns:CercaAutoreRequest"/>
125              <wsdl:output message="tns:CercaAutoreResponse"/>
126              <wsdl:fault name="UnknownWord" message="tns:UnknownWordFault"/>
127          </wsdl:operation>
```

Continueremo, in tutta questa fase, nell'approfondimento della sola operazione CercaAutore e delle sezioni ad essa collegate, poiché, appunto, il discorso per CercaTitolo e RichestaOperazioni varia solo leggermente.

In questo estratto di codice si specificano i messaggi che verranno scambiati a seconda del caso in cui ci troveremo (input, output o fault).

```
91    <wsdl:message name="CercaAutoreRequest">
92        <wsdl:part name="payload" element="typens:CercaAutore"/>
93    </wsdl:message>
94    <wsdl:message name="CercaAutoreResponse">
95        <wsdl:part name="payload" element="typens:CercaAutoreResponse"/>
96    </wsdl:message>
97    <wsdl:message name="UnknownWordFault">
98        <wsdl:part name="payload" element="typens:UnknownWordFault"/>
99    </wsdl:message>
```

Sempre a proposito dei messaggi scambiati durante l'operazione CercaAutore, in questa porzione si definisce specificatamente la loro composizione; di seguito approfondiremo gli elementi di cui sono composti.

```
30            <xsd:element name="CercaAutore">
31              <xsd:complexType>
32                  <xsd:sequence>
33                      <xsd:element name="name" type="xsd:string"/>
34                  </xsd:sequence>
35              </xsd:complexType>
36            </xsd:element>
37            <xsd:element name="CercaAutoreResponse">
38              <xsd:complexType>
39                  <xsd:sequence>
40                      <xsd:element name="name" type="xsd:string"/>
41                  </xsd:sequence>
42              </xsd:complexType>
43            </xsd:element>
44            <xsd:element name="UnknownWordFault">
45              <xsd:complexType>
46                  <xsd:sequence>
47                      <xsd:element name="word" type="xsd:string"/>
48                  </xsd:sequence>
49              </xsd:complexType>
50            </xsd:element>
```

Qui, infine, definiamo i tipi complessi utilizzati nell'operazione CercaAutore.

Passiamo, a questo punto, all'analisi del file xbean.xml:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
3         xmlns:libri="http://progetto/tesina">
4
5    <cxfbc:consumer wsdl="classpath:service.wsdl"
6                    targetService="libri:LibriService"
7                    targetInterface="libri:Libri"/>
8  </beans>
```

Nella riga 5 c'è il riferimento alla WSDL che definisce il servizio, nella 6, invece, si indica il servizio di appartenenza e, infine, nella 7 si fa riferimento all'interfaccia implementata.

5.4. Creazione e configurazione del SE

Passiamo ora all'analisi del SE del progetto, ovvero il componente che effettivamente effettuerà il servizio; utilizzeremo un componente di tipo servicemix-cxf-se creato attraverso l'archetipo servicemix-cxf-se-service-unit.

Sempre dalla cartella principale del progetto eseguiamo il seguente comando per creare la SU:

```
mvn archetype:create
-DarchetypeGroupId=org.apache.servicemix.tooling
-DarchetypeArtifactId=servicemix-cxf-se-service-unit
-DgroupId=progetto.tesina
-DartifactId=cxf-se-su
```

È stata creata la directory cxf-se-su contenente:

- un file pom.xml necessario per la compilazione dell' SU del progetto
- una directory src/main/resources con un file xbean.xml utile per la configurazione della SU
- una directory src/main/java/progetto/tesina/ contenente il file java ExampleService.java che serve per l'implementazione del nostro servizio

Inoltre Maven nel file pom.xml nella cartella principale del progetto ha aggiunto il modulo:

```
<module>cxf-se-su</module>
```

Per aumentare la leggibilità apriamo il file …/wsdl-cxf-service/cxf-se-su/pom.xml e cambiamo il nome del progetto sostituendo il campo name:

```
<name> CXF SE SU del progetto</name>
```

Dobbiamo, inoltre, specificare la versione cxf che utilizziamo aggiungendo alle properties:

```
<project>
        <properties>
                …
                <cxf-version>2.0.7</cxf-version>
                …
         </properties>
</project>
```

Infine, sempre nel file pom.xml aggiungiamo ai vari plugins il plugin org.apache.cxf :

```
<plugin>
   <groupId>org.apache.cxf</groupId>
   <artifactId>cxf-codegen-plugin</artifactId>
   <version>${cxf-version}</version>
   <executions>
      <execution>
                <phase>generate-sources</phase>
                <configuration>
                        <sourceRoot>${basedir}/target/jaxws</sourceRoot>
                        <wsdlOptions>
                           <wsdlOption>
                                <wsdl>${basedir}/src/main/resources/service.wsdl</wsdl>
                                <extraargs>
                                        <extraarg>-verbose</extraarg>
                                </extraargs>
                           </wsdlOption>
                        </wsdlOptions>
                 </configuration>
                 <goals>
                    <goal>wsdl2java</goal>
                 </goals>
```

```
        </execution>
    </executions>
</plugin>
```

Il precedente plugin serve a generare in automatico i classes java dal file WSDL attraverso wsdl2java.

Quindi il prossimo passo è copiare il file WSDL, che abbiamo modificato, e che si trova nella directory del cxf-bc-su**,** nel cxf-se-su in modo da poter generare i java classes da questo.

Copiamo quindi il file service.wsdl:

- dalla directory cxf-bc-su/src/main/resources
- alla directory cxf-se-su/src/main/resources

Ora non ci resta che implementare il nostro servizio nel file java; per fare questo sostituiamo il file ExampleService.java che è stato generato in automatico nella directory …/wsdl-cxf-service/cxf-se-su/src/main/java/progetto/tesina con il file: LibriImpl.java.

Il file LibriImpl.java dovrà realizzare il servizio e in particolare dovrà:

- far parte del package del servizio
- importare i tipi definiti nella wsdl
- fare riferimento al servizio di apparteneza e implementare l'interfaccia definita nella wsdl
- definire le 3 operazioni (CercaAutore, CercaTitolo e RichiestaOperazioni)

Ora dobbiamo configurare questa SU per farle realmente implementare il Web Service; per fare questo modificando il file chiamato xbean.xml nella direcotory src/main/resources del nostro modulo cxf-se-su inserendo:

```
        <cxfse:endpoint>
                <cxfse:pojo>
                <bean class="progetto.tesina.LibriImpl" />      </cxfse:pojo>
        </cxfse:endpoint>
```

ovvero un riferimento al POJO java che implementerà le operazioni del servizio.

5.5. Creazione e configurazione della SA

Siamo giunti all'ultima operazione da fare per completare il nostro Web Service, ovvero impacchettare i componenti nella SA in modo da poter effettuare il deploy dell'applicazione.

Dal prompt dei comandi torniamo nella directory principale del progetto e invochiamo il seguente comando per creare la SA:

```
mvn archetype:create
-DarchetypeGroupId=org.apache.servicemix.tooling
-DarchetypeArtifactId=servicemix-service-assembly
-DgroupId=progetto.tesina
-DartifactId=cxf-sa
```

È stata creata la cartella cxf-sa con al suo interno il file pom.xml; inoltre Maven nel file pom.xml nella cartella principale del progetto ha aggiunto il modulo:

```
<module>cxf-sa</module>
```

Per rendere l'output della compilazione più comprensibile, cambiamo il nome del progetto sotituendo nel pom.xml nella directory CXF-SA:

```
<name>SA del progetto</name>
```

Infine dobbiamo collegare le Service Units che abbiamo creato al Service Assembly: Maven farà questo automaticamente dopo che avremo aggiunto le corrette dipendenze nel file pom.xml della SA.

Useremo il groupId, l'artifactId e la version che troviamo nei pom.xml delle SU:

```
<dependencies>
    …
    <dependency>
        <groupId>progetto.tesina</groupId>
        <artifactId>cxf-se-su</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
    <dependency>
        <groupId>progetto.tesina</groupId>
        <artifactId>cxf-bc-su</artifactId>
```

```
            <version>1.0-SNAPSHOT</version>
        </dependency>
        …
</dependencies>
```

5.6. Build, Deploy e Test

Finalmente dopo aver creato e configurato le nostre SU e aver creato la SA e aggiunto a questa le dipendenze con le SU, siamo pronti per la compilazione del progetto.

Quindi da prompt dei comandi ci posizioniamo sulla cartella principale del progetto ed eseguiamo il comando:

```
mvn install
```

Ora, siamo pronti per il deploy del nostro Service Assembly; per far questo dobbiamo solo copiare il file cxf-sa-1.0-SNAPSHOT.jar (creato automaticamente durante la compilazione) nella directory hotdeploy di ServiceMix (dalla directory cxf-sa/target/ alla directory <ServiceMix HOME>/hotdeploy).

Ora possiamo lanciare ServiceMix da prompt dei comandi (vedere Appendice).

Per testare il progetto basta compilare ed eseguire il seguente file java:

…\client webservice\Client.java

6. Costruire un componente JMS con Apache ServiceMix

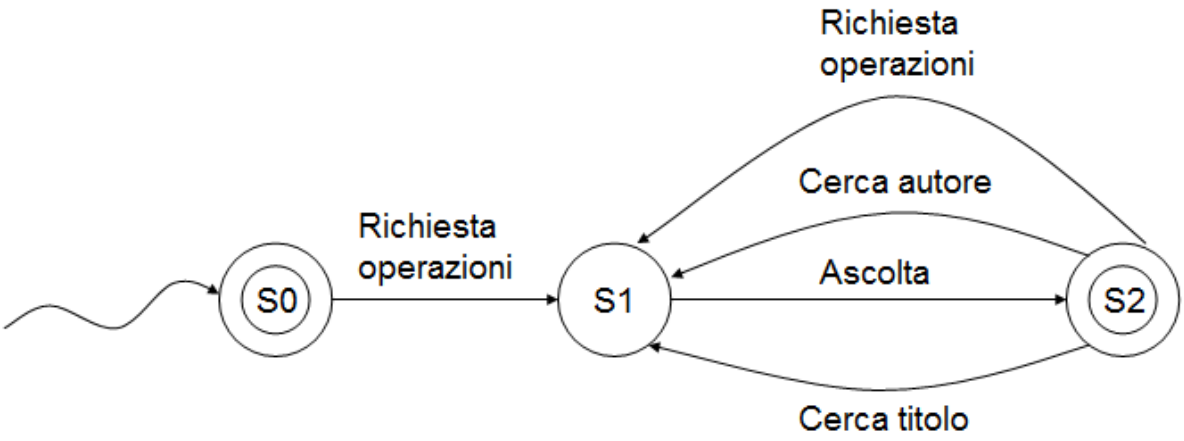
6.1. Specifica

In questo esempio implementeremo la ricerca libri gestendo le richieste tramite JMS; l'obiettivo sarà mettere in evidenza il ruolo dell'NMR nella comunicazione tra i vari componenti dell'applicazione.

Un client invierà all'ESB un messaggio per scoprire quali operazioni possono essere effettuate, l'applicazione offrirà al client la possibilità di effettuare le ricerche dei libri in due modalità, attraverso:

- L'autore
- Il titolo

Quindi il client sceglierà uno dei 2 metodi messi a sua disposizione e invierà un messaggio di richiesta. Infine l'applicazione si connetterà al database (nel nostro caso un file txt) e invierà un messaggio al client con il risultato.

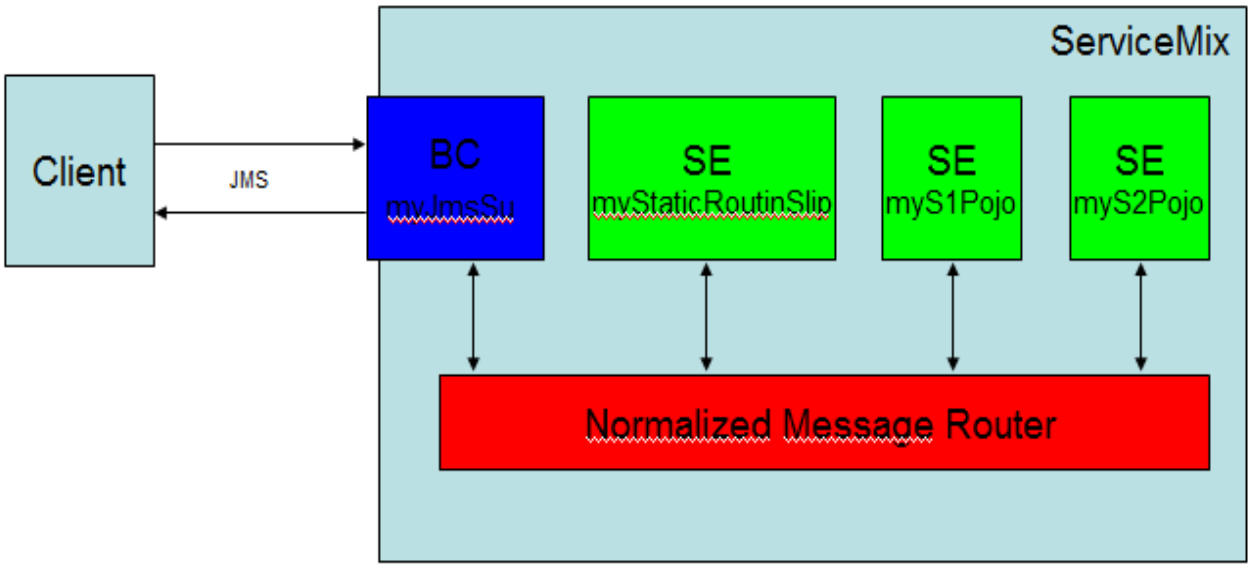


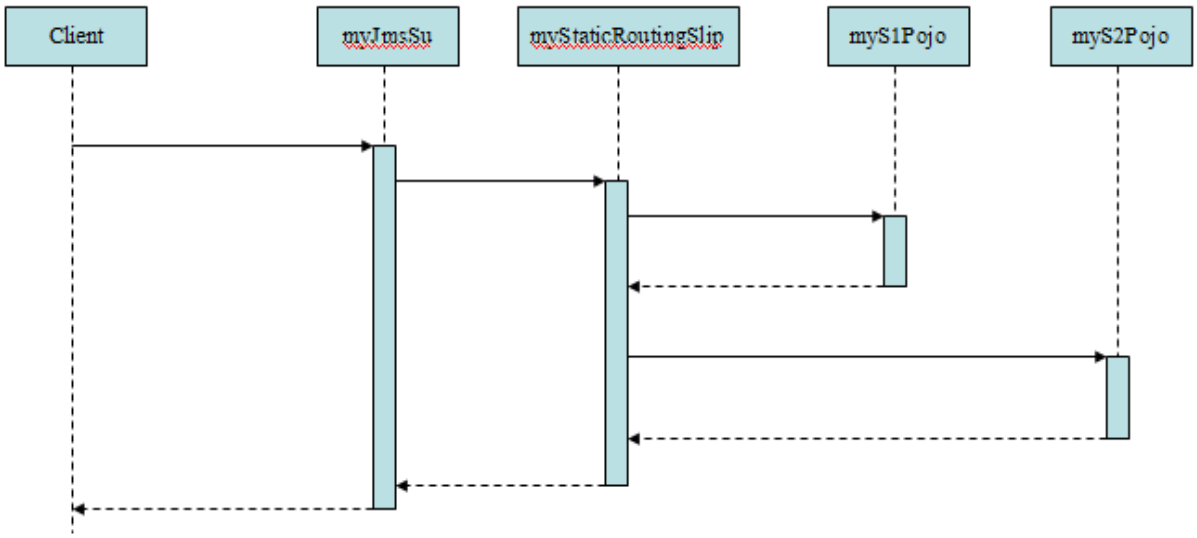
6.2. Componenti del progetto

Una JMS Service Unit (SU) che si comporta come un Binding Component (BC), mettendosi in attesa di un messaggio su una coda e occupandosi, inoltre, della traduzione necessaria per spedire il messaggio nel Normalize Message Router(NMR); per far questo useremo il componente servicemix-jms.
Una Static Routing Slip SU che controlla il flusso dei nostri messaggi nel NMR; per fare questo useremo il Service Engine (SE) servicemix-eip.
Una SU per ciascuno dei nostri due servizi S1 e S2; per questi servizi useremo una semplice implementazione POJO.

- S1 è il servizio vero e proprio, infatti fornirà i metodi *cercaAutore*, *cercaTitolo* e *richiestaOperazioni*.
- S2 è un servizio banale, che aggiungerà solo un tag all'output del servizio S1, ma che ci permetterà di vedere come funziona il routing dei messaggi attraverso l' NMR.

Infine ci sarà una Service Assembly (SA) che metterà insieme tutte queste SU in una sola unità che andrà in deploy.





Questa figura rappresenta lo scambio di messaggi tra i vari componenti.
Un client manda un messaggio su una coda JMS e aspetta la risposta su una coda JMS temporanea. Il messaggio viene trasformato dal myJmsSu in un messaggio normalizzato e inviato al componente myStaticRoutingSlip. Quest'ultimo componente lo inoltra al Service1 (S1) che svolge alcune operazioni e smista l'output al Service2 (S2) che a sua volta esegue altre operazioni.
Infine l'output di S2 viene restituito al sistema esterno tramite la coda temporanea.

6.3. Creazione del progetto

ServiceMix è fornito con alcune Maven archetypes che costruiscono automaticamente la struttura dei componenti di cui si ha bisogno nei vari progetti.

Creiamo una directory che conterrà il progetto: per comodità la chiameremo "Project_Home". Abbiamo bisogno di creare la nostra JMS SU usando l'archetype di Maven che fornisce il BC JMS servicemix-jms:

```
Project_Home> smx-arch su jms-consumer "-DgroupId=progetto.tesina2" "-DartifactId=myJmsSu"
```

Creiamo la nostra Static Routing Slip SU; per fare questo usiamo l'archetype di Maven che fornisce il SE servicemix-eip:

```
Project_Home>  smx-arch su eip "-DgroupId=progetto.tesina2" "-DartifactId=myStaticRoutingSlip"
```

Ora creiamo una SU per ciascun servizio POJO usando l'archetype di Maven per il SE servicemix-bean:

```
Project_Home> smx-arch su bean "-DgroupId=progetto.tesina2" "-DartifactId=myS1Pojo"
Project_Home> smx-arch su bean "-DgroupId=progetto.tesina2" "-DartifactId=myS2Pojo"
```

Infine dobbiamo inglobare tutte queste SU in una SA che potrà essere messa in deploy. Per creare la SA usiamo il comando:

```
Project_Home>  smx-arch sa "-DgroupId=progetto.tesina2" "-DartifactId=mySa"
```

Dopo l'esecuzione dei precedenti comandi vengono create delle cartelle che raggruppano tutti i file di ogni SU/SA.
Per configurare il comportamento dei nostri componenti JBI affinché svolgano le loro funzionalità effettive occorrerà modificare il file xbean.xml che si trova nella directory /src/main/resources di ogni nostra SU.
I file pom.xml (che si trovano nella directory principale di ogni SU), invece, servono per la compilazione delle SU del progetto.
Per fare il deploy occorrerà impacchettare tutte le SU dentro la SA; per fare questo occorre modificare il file pom.xml che si trova nella directory principale della SA.

6.4. Modifica dei file

MyJMSQueueTest

Questo è il punto di accesso e di uscita del nostro progetto in cui vengono ricevuti dei messaggi e, eventualmente, rispediti indietro; il JMS request/response verrà implementato tramite code temporanee.

Il servizio MyJMSQueueTest è inglobato in un BC SU (MyJmsSu); il file xbean.xml generato automaticamente (nella directory myJmsSu/src/main/resource) va sostituito con questo file: xbean.xml

Riportiamo il codice del file xbean.xml linkato:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
xmlns:test="http://test"
xmlns:amq="http://activemq.org/config/1.0">
<jms:endpoint service="test:MyJmsQueueTest"
        endpoint="jmsQueue"
        targetService="test:MyStaticRoutingSlipService"
        targetEndpoint="myStaticRoutingSlipSu"
        role="consumer"
        destinationStyle="queue"
        jmsProviderDestinationName="myJmsQueueTest"
        defaultMep="http://www.w3.org/2004/08/wsdl/in-out"
        connectionFactory="#connectionFactory"></jms:endpoint>
<amq:connectionFactory id="connectionFactory" brokerURL="tcp://localhost:61616" />
</beans>
```

MyStaticRoutingSlipService

La situazione che abbiamo è di un client che mette un messaggio in una coda e la BC SU MyJmsQueueTest lo estrae e lo smista verso la prima SE SU, ovvero MyStaticRoutingSlipService.

Questo servizio si occupa di smistare il messaggio al primo servizio, aspettare la risposta e quindi girarla ai servizi successivi. La sua configurazione viene definita in: myStaticRoutingSlip/src/main/resources/xbean.xml

Riportiamo il codice da sostituire nel file xbean.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:eip="http://servicemix.apache.org/eip/1.0"
    xmlns:test="http://test">
    <eip:static-routing-slip service="test:MyStaticRoutingSlipService" endpoint="myStaticRoutingSlipSu">
        <eip:targets>
            <eip:exchange-target service="test:MyS1PojoService" endpoint="myS1PojoSu"/>
            <eip:exchange-target service="test:MyS2PojoService" endpoint="myS2PojoSu"/>
        </eip:targets>
    </eip:static-routing-slip>
</beans>
```

MySnPojoService

Ora bisogna definire una SE SU per ciascun servizio a cui verranno smistati i messaggi; MyS1Pojo svolgerà effettivamente il servizio, mentre MyS2Pojo aggiungerà solo un tag per testimoniare di aver processato il messaggio.

Per configurare ciascuno dei servizi occorrerà modificare i file myS*n*Pojo/src/main/resources/xbean.xml (la *n dovrà essere* sostituita, per ogni servizio, con il rispettivo numero "1" o "2"), in questo modo:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:bean="http://servicemix.apache.org/bean/1.0"
    xmlns:test="http://test">
    <bean:endpoint service="test:MyS1PojoService" endpoint="myS1PojoSu" bean="#myBean"/>
    <bean id="myBean" class="progetto.tesina2.MyBean"/>
</beans>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:bean="http://servicemix.apache.org/bean/1.0"
    xmlns:test="http://test">
    <bean:endpoint service="test:MyS2PojoService" endpoint="myS2PojoSu" bean="#myBean"/>
    <bean id="myBean" class="progetto.tesina2.MyBean"/>
</beans>
```

I file myS1Pojo\src\main\java\progetto\tesina2\MyBean.java e myS2Pojo\src\main\java\progetto\tesina2\MyBean.java
vanno sostituiti rispettivamente con POJO1 e POJO2 che realizzeranno il servizio; in particolare dovranno:

- Far parte del package del servizio
- Fare riferimento al canale di comunicazione e implementare l'interfaccia MessageExchangeListener
- Implementare il metodo onMessage che è il metodo che viene invocato di default e svolge il servizio

Per rendere più leggibile il progetto modifichiamo i file pom.xml situati nella directory principale di ogni componente modificando il valore del tag <name>, nel seguente modo:

- myJmsSu: <name>MyJmsTest :: myJmsSu</name>
- myStaticRoutingSlip: <name>MyJmsTest :: myStaticRoutingSlip</name>
- myS1Pojo: <name>MyJmsTest :: myS1Pojo</name>
- myS2Pojo: <name>MyJmsTest :: myS2Pojo</name>
- mySa: <name>MyJmsTest :: myJmsTestSa</name>

Per ciascuno dei due pom.xml dei servizi POJO trovare le seguenti righe di codice:

```
<properties>
        <servicemix-version>3.2.1</servicemix-version>
</properties>
```

e aggiungere subito dopo:

```
<servicemix-version>3.2.1</servicemix-version>
```

la seguente riga:

```
<componentName>servicemix-bean</componentName>
```

Il file pom.xml della SA va ulteriormente modificato aggiungendo le dipendenze a tutte le SU del progetto. Quindi alla dipendenza da junit (generata di default) vanno aggiunte le seguenti dipendenze:

```
<dependencies>
        …
        <dependency>
                <groupId>progetto.tesina2</groupId>
                <artifactId>myJmsSu</artifactId>
                <version>1.0-SNAPSHOT</version>
        </dependency>
        <dependency>
                  <groupId>progetto.tesina2</groupId>
                  <artifactId>myStaticRoutingSlip</artifactId>
                  <version>1.0-SNAPSHOT</version>
        </dependency>
        <dependency>
                  <groupId>progetto.tesina2</groupId>
                  <artifactId>myS1Pojo</artifactId>
                 <version>1.0-SNAPSHOT</version>
```

```
        </dependency>
        <dependency>
                <groupId>progetto.tesina2</groupId>
                <artifactId>myS2Pojo</artifactId>
                <version>1.0-SNAPSHOT</version>
        </dependency>
        ....
</dependencies>
```

6.5. Build, Deploy e Test

Ora siamo pronti per il build delle nostre SU. Da linea di comando entrare nella directory principale di ogni SU (myJmsSu, myStaticRoutingSlip, myS1Pojo, myS2Pojo) ed eseguire il comando: mvn install

Infine possiamo fare il build della nostra SA accedendo nella direcotory principale della SA ed eseguendo il comando: mvn install

Per effettuare il deploy del progetto basta copiare il file mySa-1.0-SNAPSHOT.jar presente nella directory /target della SA nella directory hotdeploy di ServiceMix.

A questo punto possiamo lanciare ServiceMix eseguendo il comando: servicemix

Per testare il progetto creiamo un client che metterà un messaggio nella coda JMS e attenderà su una coda temporanea il risultato.

Nel client andranno integrate le librerie di ServiceMix per la gestione di JMS.

Il client può essere lanciato eseguendo il file tesinaSM.jar

Appendice

A. Installazione e configurazione di ServiceMix

Requisiti:

JDK 6 (reperibile qui http://java.sun.com/javase/downloads/index.jsp)

ServiceMix 3.2.1 (reperibile qui http://servicemix.apache.org/servicemix-321.html)

Maven 2.0.9 (reperibile qui: http://maven.apache.org/download.html)

Installazione:

L'installazione di ServiceMix e Maven consiste nella semplice estrazione dell'archivio scaricato.

Per semplificare la descrizione delle operazioni successive indicheremo con <ServiceMix_home> la directory in cui è stato estratto l'archivio di ServiceMix e <Maven_home> la directory in cui è stato estratto l'archivio di Maven.

Configurazione:

Per poter usare correttamente ServiceMix occorre, dopo l'installazione, configurare alcune variabili di ambiente.

Occorre, prima di tutto, verificare se è presente la variabile JAVA_HOME; qualora non sia presente occorre crearla e settarla con il percorso in cui è installata la JDK.

Successivamente vanno aggiunte nella variabile di ambiente Path i seguenti percorsi:

- <ServiceMix_home>/bin
- <Maven_home>/bin

Generazione e compilazione degli archetypes di Maven:

Maven mette a disposizione una serie di archetypes: gli archetypes non sono altro che componenti base predefiniti utili per semplificare lo sviluppo dell'applicazione.

Per generare un archetype sono disponibili due metodologie:

1. tramite il comando mvn archetype:create (più le particolare opzioni legate al singolo progetto e alla particolare archetype che si vuole creare)

   es.:

   ```
   mvn archetype:create
   -DarchetypeGroupId=org.apache.servicemix.tooling
   -DarchetypeArtifactId=servicemix-cxf-bc-service-unit
   -DgroupId=progetto.tesina
   -DartifactId=cxf-bc-su
   ```

2. tramite il comando smx-arch (più le particolare opzioni legate al singolo progetto e alla particolare archetype che si vuole creare)

   es.:

   ```
   smx-arch su jms-consumer
   "-DgroupId=progetto.tesina2"
   "-DartifactId=myJmsSu"
   ```

Una volta effettuate le modifiche desiderate all'archetype occorrerà compilarlo; per far questo occorre, da console, eseguire il comando mvn install dalla home directory dell'archetype.

Deploy di una applicazione:

Per effettuare il deploy di una applicazione, basta copiare l'archivio JAR presente nella directory target della SA dell'applicazione nella directory <ServiceMix_home>/hotdeploy.

Esecuzione:

Per lanciare ServiceMix basta lanciare da console il comando: servicemix.