Aplicații Integrate pentru Întreprinderi

Laborator 12

13.01.2014

Comunicații Asincrone folosind Java Message Service

Scopul laboratorului îl reprezintă dezvoltarea unei aplicații conținând mai multe componente care comunică asincron prin intermediul specificației JMS (Java Message Service) şi al unui server de aplicații.

1. Java Message Service – prezentare generală

Un sistem de gestiune al mesajelor este un mecanism de comunicare între componente ale unei aplicații. De regulă, este un sistem punct la punct (*eng.* peer-to-peer) astfel încât un client poate primi mesaje şi poate transmite mesaje de la şi către orice alt client. Fiecare client se conectează la un agent responsabil cu gestiunea mesajelor ce oferă funcționalități pentru crearea lor, transferul prin infrastructura de comunicație şi realizarea diferitelor operații ce le implică.

Prin intermediul unui sistem de gestiune al mesajelor, se permite dezvoltarea de aplicații distribuite **slab cuplate**, astfel încât o componentă transmite un mesaj către o adresă în timp ce altă componentă primeşte un mesaj de la adresa respectivă, aceasta fiind singurul element pe care îl au în comun, comunicația nefiind condiționată de disponibilitatea simultană a acestora. Mai mult, componenta care transmite mesajul nu trebuie să ştie nimic despre componenta care îl primeşte şi nici invers. Acestea trebuie să cunoască numai formatul pe care trebuie să îl respecte mesaje precum şi adresa la care, respectiv de la care să fie transferate[1].

Java Message Service (JMS) este un API Java care permite aplicațiilor crearea de mesaje, transferul lor prin infrastructura de comunicație (operații de transmitere şi primire) şi prelucrarea acestora. Sunt puse la dispoziție şi un set de interfețe cu semantica asociată ce permit programelor scrise folosind limbajul de programare Java să comunice cu alte implementări ale unor sisteme pentru gestiunea mesajelor. De asemenea, este redus sistemul de concepte pe care programatorul trebuie să şi le însuşească astfel încât să poată dezvolta produse care implică transferul de mesaje, oferind suficiente funcționalități pentru realizarea de aplicații complexe. Se încearcă şi asigurarea portabilității aplicațiilor care utilizează JMS între diferiți producători care implementează acest API.

[1] Din acest punct de vedere, sistemul de transmitere al mesajelor se deosebeşte de tehnologiile strâns cuplate (cum ar fi comunicațiile la distanță de tip RPC) care solicită aplicației să cunoască funcționalitățile puse la dispoziție precum şi modul în care acestea pot fi accesate. Totodată, sistemul de transmitere al mesajelor se deosebeşte şi de sistemul de poştă electronică, metodă ce presupune interacțiunea între persoane (sau între aplicații şi persoane), întrucât acesta implică comunicația între componente ale unor aplicații.

Comunicația în cazul Java Message Service este caracterizată, pe lângă slaba cuplare a componentelor ce interacționează şi prin:

❶ **asincronicitate**: o componentă nu trebuie să primească în mod necesar mesajele la acelaşi moment în care acestea au fost transmise; o aplicație poate transmite mesajele, realizând ulterior alte operații în timp ce programul care trebuie să le primească poate îndeplini alte sarcini înainte de a le primi;

❷ **siguranța transferului** întrucât furnizorul serviciilor de gestiune a mesajelor garantează că un mesaj este transmis în mod cert o singură dată[2].

În prezent, versiunea specificației JMS este 2.0.

Este probabil ca un dezvoltator de aplicații integrate pentru întreprinderi să aleagă un sistem de gestiune al mesajelor în detrimentul unor componente strâns cuplate în situația în care doreşte ca funcționalitatea modulelor să nu depindă de alte informații puse la dispoziție de interfețele altor aplicații (astfel încât procesul de întreținere să se poată realiza cu uşurință, înlocuindu-se anumite componente şi păstrând pe altele), ca programul să funcționeze şi atunci când unele module nu rulează simultan sau dacă fluxul operațional permite ca ulterior transmiterii unor mesaje componentele pot opera indiferent dacă au primit sau nu un răspuns.

Într-o aplicație integrată pentru întreprinderi, un sistem de mesagerie poate fi utilizat pentru interacțiunea dintre componente, astfel că unele operații să poată fi realizate automat prin prelucrarea mesajelor de îndată ce modulul căruia îi sunt adresate devine disponibil.

Scopul Java Message Service, atunci când a fost dezvoltat, a constat în permiterea aplicațiilor Java de a accesa sistemele orientate pe mesaje existente (*eng.* Middleware Message-Oriented Systems). De atunci, distribuitorii au adoptat şi implementat acest model, astfel că un produs de acest tip oferă funcționalități complete de mesagerie pentru o organizație. Acesta este integrat în Java Enterprise Edition, astfel încât poate fi utilizat pentru transfer de mesaje în cadrul acestor componente[3]. Astfel, JMS a contribuit la îmbunătățirea altor părți din cadrul platformei Java EE, simplificând procesul de dezvoltare al aplicațiilor, facilitând interacțiunea dintre programele dezvoltate utilizând această tehnologie şi sistemele moştenite responsabile cu gestiunea mesajelor[4]. Furnizorul JMS poate fi integrat cu un server de aplicații folosind arhitectura Java EE Connector, accesul realizându-se printr-un adaptor de resuse, astfel că se permite interacțiunea între distribuțiile JMS şi diferite servere de aplicații.

---

[2] Sunt disponibile în acelaşi timp şi alte nivele de siguranță a transferului în cazul în care aplicațiile nu sunt afectate de pierderea unor mesaje, respectiv de primirea unor duplicate.

[3] Aplicațiile client, componentele Enterprise JavaBeans (EJB) şi componentele web pot trimite sau primi asincron un mesaj. Mai mult, aplicațiile client pot indica un ascultător de mesaje care permit ca mesajele să fie livrate asincron prin intermediul unei notificări atunci când acesta devine disponibil. Componentele orientate pe mesaje (*eng.* message-driven bean) oferă posibilitatea consumării mesajelor în cadrul unui container EJB, întrucât serverul de aplicații le grupează pentru a oferi prelucrarea concurentă a mesajelor. De asemenea, operațiile de trimitere şi primire a mesajelor pot participa la o tranzacție Java Transaction API (JTA), care permite ca operațiile JMS şi accesul la baza de date să se realizeze în mod atomic.

[4] Un programator poate implementa cu uşurință funcționalități noi unei aplicații Java EE pentru care au fost definite evenimente legate de fluxurile operaționale şi procesele de afaceri prin dezvoltarea unei componente orinetate pe mesaje care le poate procesa. Şi platforma Java EE îmbunătățeşte API-ul JMS prin suportul pentru tranzacții JTA şi consumarea concurentă a mesajelor.

2. Mesaje: categorii şi gestiunea lor în JMS

Scopul unei aplicații JMS îl reprezintă producerea şi consumarea de mesaje care să poată fi utilizate de diferite componente. Mesajele definite de API-ul JMS au un format flexibil care permite specificarea unei structuri compatibile diverselor programe ce rulează pe diferite platforme.

Un mesaj JMS (definit de clasa `javax.jms.Message`) are **trei părți**: un antet (obligatoriu), proprietăți şi un conținut.

Antetul unui mesaj JMS conține un număr de câmpuri predefinite care conțin valori utilizate atât de clienți cât şi de furnizorii serviciului de mesagerie pentru a identifica mesajele şi pentru a le direcționa către componentele dorite. Fiecare atribut din cadrul antetului are definite metode de tip setter şi getter. Pentru unele, valorile sunt stabilite de către client, însă pentru cele mai multe valorile sunt indicate în mod automat de furnizorul serviciului de mesagerie (prin intermediul metodei `send`), suprascriind datele specificate.

Atribut	Descriere	Componenta care îi stabileşte valoarea
`JMSDestination`	destinația la care este transmis mesajul	metoda `send` a furnizorului de servicii de mesagerie
`JMSDeliveryMode`	mecanismul de transfer specificat atunci când este transmis mesajul	metoda `send` a furnizorului de servicii de mesagerie
`JMSDeliveryTime`	momentul de timp la care a fost transmis mesajul la care se adaugă întârzierea pentru livrare indicată la transmiterea mesajului	metoda `send` a furnizorului de servicii de mesagerie
`JMSExpiration`	perioada de expirare a mesajului	metoda `send` a furnizorului de servicii de mesagerie
`JMSPriority`	prioritatea mesajului	metoda `send` a furnizorului de servicii de mesagerie
`JMSMessageID`	valoare care identifică în mod unic fiecare mesaj transmis de furnizorul de servicii de mesagerie	metoda `send` a furnizorului de servicii de mesagerie
`JMSTimestamp`	momentul de timp la care mesajul a fost livrat furnizorului de servicii de mesagerie pentru a fi transmis	metoda `send` a furnizorului de servicii de mesagerie
`JMSCorrelationID`	valoare prin care un mesaj referă un altul prin indicarea atributului `JMSMessageID`	aplicația client
`JMSReplyTo`	locația la care ar trebui transmise răspunsurile la mesaj	aplicația client
`JMSType`	identificatorul cu privire la tip furnizate de aplicația client	aplicația client
`JMSRedelivered`	valoare care indică dacă mesajul este retransmis	furnizorul de servicii de mesagerie (înainte de transmitere)

Utilizatorul poate specifica un **set de proprietăți** pentru mesaje dacă sunt necesare atribute suplimentate pentru care să se specifice valori în plus față de cele oferite. Acestea pot fi utilizate pentru a oferi compatibilitate cu alte sisteme de mesagerie sau pot fi folosite pentru a folosi selectoare de mesaje.

API-ul JMS oferă şi unele nume de proprietăți predefinite, prefixate de `JMSX`. Un furnizor de servicii de mesagerie trebuie să implementeze doar una dintre acestea, `JMSXDeliveryCount` (care indică de câte ori a fost transmis un mesaj), restul fiind opționale.

Folosirea proprietăților predefinite sau a celor definite de utilizator este doar opțională.

În cadrul API-ului JMS sunt definite mai multe **tipuri de mesaje**, fiecare având **un conținut** specific, permițând prelucrarea datelor în formaturi diferite.

Tip de Mesaj	Conținut
`TextMessage`	un obiect `java.lang.String`
`MapMessage`	un set de perechi (atribut, valoare), atributul fiind obiect de tip `java.lang.String` iar valoarea un tip primitiv de date; valorile pot fi accesate fie secvențial printr-un iterator / enumerator (ordinea nefiind definită), fie aleator prin denumirea atributului
`BytesMessage`	un flux de octeți neinterpretați; acest tip de mesaj este folosit pentru codificarea unui conținut astfel încât să corespundă unui format de mesaj existent
`StreamMessage`	un flux de valori primitive, specificate şi accesate secvențial
`ObjectMessage`	un obiect serializabil (ce implementează `java.io.Serializable`) definit în limbajul de programare Java
`Message`	vid, mesajul fiind compus doar din antet şi proprietăți, acesta fiind util atunci când se doreşte semnalizarea unui eveniment[5]

Sunt oferite metode pentru construirea de mesaje de fiecare tip, stabilindu-se conținutul specific. Un obiect de tip `TextMessage` poate fi creat astfel:

```
TextMessage message = context.createTextMessage();
message.setText(content);
context.createProducer().send(message);
```

Mesajul va fi transmis ca un obiect generic de tip `Message`, acesta trebuind convertit la tipul corespunzător, folosindu-se metode specifice[6] pentru accesarea conținutului (respectiv a antetelor şi a proprietăților, dacă este necesar).

```
Message message1 = consumer.receive();
MapMessage mapMessage = (MapMessage)message1;
Enumeration<String> mapAttributes = mapMessage.getMapNames();
for(String mapAttribute: mapAttributes)
    String mapValue = mapMessage.getObject(mapAttribute);
Message message2 = consumer.receive();
ObjectMessage objectMessage = (ObjectMessage)message2;
CustomMessage customMessage = (CustomMessage)objectMessage.getObject();
```

Alternativ, se poate apela metoda `getBody` a clasei `Message`, indicându-se tipul de mesaj ca argument.

```
Message message = consumer.receive();
if (message instanceof TextMessage) {
    String content = message.getBody(String.class);
}
```

[5] Apelul `context.createProducer().send(receiver, context.createMessage());` transmite un astfel de mesaj. O situație în care se transmite un astfel de mesaj este cea în care se semnalează că toate mesajele au fost transmise.

[6] Spre exemplu, se pot folosi metodele de citire orientate pe fluxuri oferite de tipul de mesaj `BytesMessage`. Pentru a se obține conținutul unui mesaj de tip `StreamMessage`, trebuie realizată întotdeauna operația de convertire.

API-ul JMS oferă totodată posibilitatea de a crea şi de a primi rapid mesaje de tipul `TextMessage`, `BytesMessage`, `MapMessage` sau `ObjectMessage`, indicând conținutul de transmis direct în metoda `send`, respectiv prin utilizarea metodei `receiveBody`[7]:

```
context.createProducer().send(receiver, content);
String content = consumer.receiveBody(String.class);
```

3. Arhitectura JMS

O aplicație JMS este formată din următoarele componente:

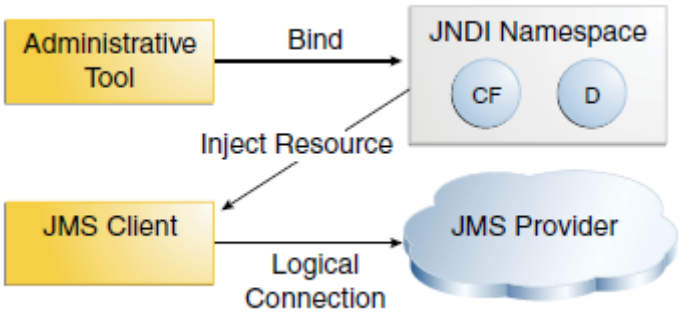
❶ **furnizorul JMS** este un sistem de mesagerie ce implementează interfețele specificate de API, oferind totodată funcționalități administrative şi de control; o implementare a platformei Java EE care suportă întregul profil incluse şi un furnizor JMS; de regulă, acesta este reprezentat de un server de aplicații;

❷ **clienții JMS** sunt aplicații sau componente scrise în limbajul de programare Java ce produc şi consumă mesaje; atât programele Java EE cât şi cele Java SE pot implementa un astfel de comportament;

❸ **mesajele** sunt obiecte care comunică informații între clienții JMS;

❹ **obiectele administrate** sunt configurate pentru a fi utilizate de clienții JMS; există două tipuri de obiecte administrate şi anume fabrici de conexiuni (*eng.* connection factories) şi destinații (*eng.* destinations); administratorul poate crea obiecte care sunt disponibile tuturor aplicațiilor care folosesc o instanță a unui server de aplicații[8].

Interacțiunea dintre componente constă în faptul că obiectele administrate (fabricile de conexiuni şi fabricile de destinații) se asociază unui spațiu de nume JNDI (Java Naming and Directory Interface)[9]. Un client JMS poate utiliza injectarea resurselor (*eng.* resource injection) spre a accesa obiectele administrate stabilind o conexiune logică către acestea prin intermediul furnizorului de servicii de mesagerie.



Arhitectura JMS

[7] Metoda `receiveBody` poate fi utilizată pentru a primi orice tip de mesaj cu excepția `StreamMessage` şi `Message`, atâta timp când conținutul mesajelor poate fi atribuit unui anumit tip de date.

[8] Există şi posibilitatea de a se utiliza adnotări pentru a se crea obiecte care sunt specifice unei anumite aplicații.

[9] Prin intermediul JNDI se permite aplicațiilor distribuite să caute servicii într-un mod abstract, independent de resurse. Astfel, în cazul în care se schimbă mediul de producție al unei aplicații, aceasta nu trebuie modificată pentru a putea rula, accesând nişte resurse la distanță. Totodată, sunt ascunse detalii pe care un utilizator ce foloseşte resursa nu ar trebui să le cunoască, acestea fiind configurate la nivelul serverului de aplicații.

Înainte de dezvoltarea JMS, cele mai multe produse ce implementau sisteme de mesagerie suportau mecanisme de tipul punct la punct respectiv publicare-abonare (*eng.* publish-subscribe). Un furnizor JMS trebuie să ofere ambele stiluri de comunicare, punând la dispoziția programatorilor interfețe specifice pentru fiecare dintre acestea.

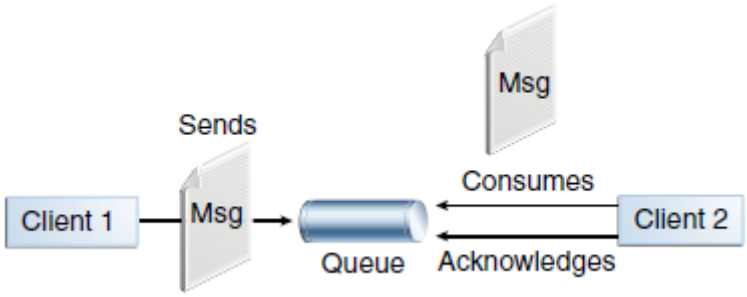
Prin intermediul API-ului JMS, programatorii nu sunt limitați la folosirea unui anumit mecanism exclusiv, permițându-se folosirea aceluiaşi cod sursă pentru transmiterea şi primirea de mesaje folosind fie modelul punct la punct, fie modelul publicare-abonare. Cu toate că destinațiile unui mesaj sunt specifice stilului de comunicare folosit, comportamentul aplicației depinzând parțial de utilizarea unei cozi (*eng.* queue) respectiv a unei teme (*eng.* topic), codul sursă propriu-zis este comun celor două mecanisme, dând aplicației flexibilitate întrucât aceasta poate fi reutilizată uşor.

O aplicație **punct la punct** este construită pe conceptul de **cozi**, destinatari şi expeditori. Fiecare mesaj este transmis de un destinatar către o anumită coadă, de unde expeditorii le primesc. Cozile reține toate mesajele transmise către ele până când acestea sunt primite sau până când expiră.

Caracteristicile unui sistem de mesagerie punct la punct sunt:

❶ fiecare mesaj are un singur consumator;

❷ destinatarul va primi mesajul indiferent de disponibilitatea sa la momentul de timp la care expeditorul a transmis mesajul.



Modelul de comunicare punct la punct

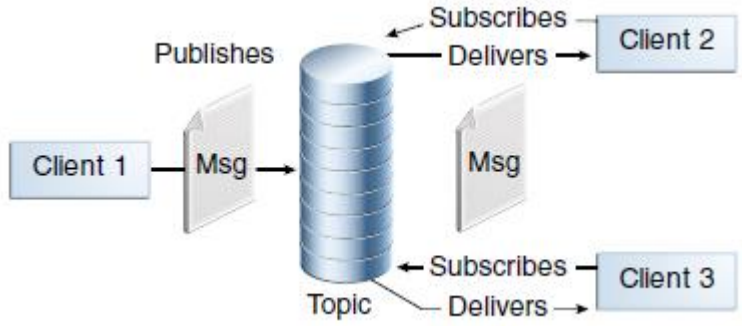
Utilizarea mecanismului de comunicare punct la punct este limitată la situația în care se doreşte ca fiecare mesaj să fie prelucrat de un singur client.

Într-o componentă ce respectă modelul **publicare-abonare**, clienții transmit mesajele către o **temă**, putând realiza operații de publicare sau abonare în mod dinamic. Responsabilitatea pentru distribuirea mesajelor de la clienții care publică mesaje către abonați revine sistemului de mesagerie. Tema va reține mesajele numai în răstimpul necesar pentru a le transfera.

În cazul acestui model, trebuie făcută distincția între abonatul propriu-zis şi abonamentul pe care acesta îl creează: dacă abonatul este un obiect JMS de tip consumator în cadrul unei aplicații, abonamentul este o entitate care există la nivelul furnizorului de servicii de mesagerie. O temă poate avea mai mulți abonați, însă unui abonament îi corespunde, de regulă, un singur client[10].

[10] Există de asemenea posibilitatea de a se crea abonamente partajate, identificate printr-un anumit şir de caractere, care pot fi folosite de mai mulți clienți concomitent însă acestea sunt folosite pentru distribuirea prelucrărilor întrucât un mesaj nu poate fi primit decât de către un singur consumator.

Caracteristicile unui sistem publicare-abonare sunt:
❶ fiecare mesaj poate avea mai mulți consumatori;
❷ un client care se abonează la o temă poate consuma doar mesajele transmise după ce a creat abonamentul, el trebuind să fie disponibil pentru a putea consuma în continuare mesaje[11].



Modelul de comunicare publicare-abonare

Utilizarea mecanismului de comunicare publicare-abonare este adecvată situațiilor în care un mesaj poate fi procesat de orice număr de consumatori (inclusiv nici unul).

Cele mai multe produse ce oferă facilități de transfer al mesajelor sunt asincrone, în sensul că nu există o dependență temporală fundamentală între momentul la care a fost produs mesajul şi momentul la care acesta a fost consumat. Totuşi, specificația JMS permite faptul ca un mesaj să fie procesat în ambele moduri:
❶ **sincron**, consumatorul primind mesajul în mod explicit de la destinatar printr-un apel al metodei `receive`[12];
❷ **asincron**, în care aplicația client va defini un ascultător de mesaje, a cărui funcționare este similară ca în cazul evenimentelor: de fiecare dată când se transmite un mesaj, furnizorul de servicii de mesagerie va face ca acesta să fie primit prin apelul metodei `onMessage`, ce acționează asupra conținutului[13].

4. Modelul de programare al API-ului JMS

Principalele componente ale unei aplicații JMS sunt obiectele administrate (fabrici de conexiuni şi destinații), conexiuni şi sesiuni (reținute împreună într-un obiect de tipul `JMSContext`), producătorii şi consumatorii de mesaje precum şi mesajele propriu-zise[14].

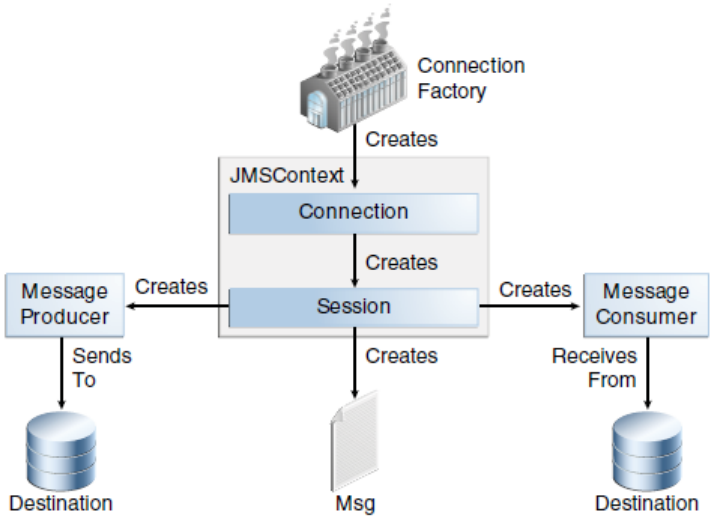
---

[11] O astfel de cerință este însă relaxată de API-ul JMS într-o anumită măsură, permițând clienților să creeze abonamente durabile, care păstrează mesajele transmise în perioada în care consumatorii nu sunt activi. Subscripțiile de acest tip oferă flexibilitate şi asigură siguranța transmiterii mesajelor pe care le oferă cozile, permițând în acelaşi timp transmiterea de mesaje către mai mulți destinatari.

[12] Metode `receive` se poate bloca până la momentul în care un mesaj este transmis sau până când a expirat o anumită perioadă de aşteptare.

[13] În cazul unei aplicații Java EE, o componentă orientată pe mesaje funcționează pe post de ascultător de mesaje (având definită de asemenea metoda onMessage), cu diferența că aceasta nu trebuie înregistrată cu un anumit consumator.

[14] O fabrică de conexiuni poate fi folosită pentru crearea unui obiect de tipul `JMSContext` (inițial este creat obiectul de tip Connection şi din acesta obiectul de tip Session). Acesta este folosit pentru crearea mesajului propriu-zis, dar şi a producătorului de mesaje care transmite mesaje către destinație precum şi a consumatorului de mesaje care primeşte mesaje de la destinație.



Modelul de programare al API-ului JMS

În cadrul unei aplicații JMS, obiectele administrate – fabricile de conexiuni şi destinațiile – sunt întreținute administrativ[15] mai degrabă decât programatic întrucât tehnologia prin intermediul căruia sunt implementate poate diferi de la o implementare la alta a API-ului.

Clienții JMS accesează obiectele administrate prin intermediul unor interfețe portabile, astfel încât se poate face trecerea între diferite distribuții fără prea multe modificări.

Configurarea lor se face în contextul unui spațiu de nume JNDI[16], astfel că accesul la nivelul clienților se face prin injectarea resurselor.

O **fabrică de conexiuni** este un obiect utilizat de client pentru a crea o conexiune către un furnizor JMS de servicii de mesagerie. Un astfel de obiect înglobează un set de parametrii ce conțin configurări ale conexiunii, aceştia fiind definiți de administrator. O fabrică de conexiuni este o instanță a uneia din interfețele `ConnectionFactory`, `QueueConnectionFactory`, `TopicConnectionFactory`. Fiecare aplicație client JMS începe cu injectarea unei fabrici de conexiuni într-un obiect `ConnectionFactory`[17].

```
queueConnectionFactory = (QueueConnectionFactory)
               initialContext.lookup(Constants.QUEUE_CONNECTION_FACTORY_NAME);
topicConnectionFactory = (TopicConnectionFactory)
               initialContext.lookup(Constants.TOPIC_CONNECTION_FACTORY_NAME);
```

Obiectul de tip `InitialContext` este creat pe baza unor parametrii care indică mecanismul de conectare la serverul de aplicații, inclusiv protocolul de comunicații, adresa şi portul.

---

[15] Gestiunea obiectelor administrate face parte din cadrul sarcinilor administrative ce pot varia de la un distribuitor la altul.

[16] Specificația platformei Java EE permite unui programator crearea de obiecte administrate folosind adnotări sau elemente descriptive în cadrul procesului de dezvoltare (*eng.* deployment descriptor elements). Astfel de obiecte vor fi însă specifice aplicației pentru care au fost create. Definițiile ale unui element descriptiv în cadrul procesului de dezvoltare le suprascriu pe cele indicate de adnotări.

[17] În cazul unei aplicații Java EE, fabrica de conexiuni va fi indicată prin intermediul numelui logic JNDI (`java:comp/DefaultJMSConnectionFactory`, pentru obiectul implicit preconfigurat pe serverul de aplicații), folosindu-se adnotarea `@Resource` cu parametrul `lookup`:

```
@Resource(lookup = "java:comp/DefaultJMSConnectionFactory ")
private static ConnectionFactory connectionFactory;
```

O **destinație** este un obiect folosit de client pentru a specifica destinația mesajelor pe care pe produce şi sursa mesajelor pe care le consumă. În mecanismul de comunicare punct la punct, destinațiile sunt denumite cozi. Pentru modul de comunicare publicare-abonare, destinațiile poartă numele de teme. O aplicație JMS poate folosi mai multe cozi, mai multe teme sau obiecte din ambele categorii.

Pentru a crea o destinație, trebuie specificată o resursă JMS care specifică numele JNDI pentru destinație. Fiecare resursă de tip acest tip referă o anumită locație fizică[18].

Pe lângă injectarea unei fabrici de conexiuni în aplicația client, de regulă se injectează şi o resursă de tip destinație. Spre diferență însă de fabricile de conexiuni, destinațiile sunt specifice modelului de comunicare punct la punct sau publicare-abonare. Pentru a crea o aplicație care permite folosirea aceluiaşi cod sursă atât pentru cozi cât şi pentru teme, se va folosi un obiect de tipul `Destination`[19].

```
queue = (Queue) initialContext.lookup(Constants.QUEUE_NAME);
topic = (Topic) initialContext.lookup(Constants.TOPIC_NAME);
```

În cadrul interfețelor de bază, se pot combina diferitele fabrici de conexiuni cu tipuri de destinații diferite[20].

O **conexiune** încapsulează o conexiune virtuală către un furnizor JMS de servicii de mesagerie[21]. O conexiune poate fi utilizată pentru a crea una sau mai multe sesiuni[22]. De regulă, o conexiune este creată printr-un obiect `JMSContext`, dar se pot utiliza şi obiecte `Connection`, `QueueConnection`, `TopicConnection`.

```
QueueConnection queueConnection = queueConnectionFactory.createQueueConnection();
TopicConnection topicConnection = topicConnectionFactory.createTopicConnection();
```

O **sesiune** reprezintă un context ce rulează într-un singur fir de execuție pentru producerea şi consumarea de mesaje. Deşi sesiunea (împreună cu conexiunea) sunt create printr-un obiect `JMSContext`, se pot utiliza şi obiecte `Session`, `QueueSession` şi `TopicSession`.

```
QueueSession queueSession =
             queueConnection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
TopicSession topicSession =
             topicConnection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
```

---

[18] Un administrator poate să creeze o locație fizică în mod explicit, însă dacă nu o va face, serverul de aplicații va realiza această operație atunci când este necesar, ştergând resursa dacă destinația este înlăturată.

[19] În cazul unei aplicații Java EE, obiectele administrate sun de obicei plasate în subcontextul de nume `jms`. Injectarea resurselor în acest caz se face tot prin intermediul adnotării `@Resource`, folosind atributul `lookup`:

```
@Resource(lookup = "jms/MyQueue")
private static Queue queue;
@Resource(lookup = "jms/MyTopic")
private static Topic topic;
```

[20] Astfel, o resursă de tip `QueueConnectionFactory` se poate folosi în corelație cu un obiect `Topic` în timp ce o resursă de tip `TopicConnectionFactory` poate fi utilizată împreună cu un obiect `Queue`.

[21] Procesul de descoperire a resurselor definite în contextul JNDI se face prin intermediul protocolului IIOP (comunicația având loc pe portul 3700) în timp ce transferul propriu-zis al mesajelor se face printr-un protocol specific JMS, pe portul 7676.

[22] În cazul platformei Java EE, posibilitatea de a crea mai multe sesiuni din cadrul unei singure conexiuni este limitată doar pentru aplicațiile client. În cazul aplicațiilor de tip enterprise şi web, o conexiune nu poate crea mai mult de o sesiune.

Sesiunile sunt utilizate spre a crea producători şi consumatori de mesage, mesaje, obiecte pentru consultarea (*eng.* browser) cozilor şi destinații temporare. Sesiunile serializează execuția ascultătorilor de mesaje. O sesiune oferă un context tranzacțional cu care grupează un set de operații de transmitere şi de primire într-o unitate atomică.

Un obiect `JMSContext` asociază o conexiune şi o sesiune într-un singur obiect, oferind atât o conexiune activă către furnizorul JMS de servicii de mesagerie cât şi un context ce rulează într-un singur fir de execuție pentru operații de transmitere şi primire de mesaje[23].

Un astfel de obiect se creează de obicei într-un bloc `try`-cu-resurse[24], printr-un apel al metodei `createContext` pe un obiect de tip fabrică de conexiuni:

```
JMSConext context = connectionFactory.createContext();
```

Dacă este apelată fără argumente din contextul unei aplicații client sau dintr-un client Java SE precum şi din cadrul unui container Java EE EJB / web în care nu există tranzacții JTA în progres la momentul respectiv de timp, metoda `createContext` realizează o sesiune fără tranzacții în care confirmarea pentru primirea mesajelor se face automat (`JMSContext.AUTO_ACKNOWLEDGE`)[25]. Apelată dintr-un container Java EE în care există tranzacții JTA în progres la momentul de timp respectiv, va determina crearea unei sesiuni cu tranzacții. Acest tip de comportament poate fi obținut în cadrul aplicațiilor client sau al clienților Java SE prin precizarea explicită a parametrului `JMSContext.SESSION_TRANSACTED`. Sesiunea foloseşte întotdeauna tranzacții locale.

Un **producător de mesaje** este un obiect creat prin intermediul unui obiect `JMSContext` sau a unei sesiuni şi utilizat pentru transmiterea de mesaje către o destinație. Un astfel de obiect implementează interfața `JMSProducer` şi este creat printr-un apel al metodei `createProducer`.

```
try (JMSContext context = connectionFactory.createContext();) {
    JMSProducer producer = context.createProducer();
    // ...
}
```

Întrucât obiectul `JMSProducer` consumă destul de puține resurse, nu este necesar ca acesta să fie reținut într-o variabilă distinctă, putându-l crea de fiecare dată când este transmis un mesaj, prin intermediul metodei send.

```
context.createProducer().send(destination, message);
```

Un **consumator de mesaje** este un obiect creat prin intermediul unui obiect `JMSContext` sau a unei sesiuni şi utilizat pentru primirea de mesaje de la o destinație. Un astfel de obiect implementează interfața `JMSConsumer` şi este creat printr-un apel al metodei `createConsumer`:

```
try (JMSContext context = connectionFactory.createContext();) {
    JMSConsumer consumer = context.createConsumer(destination);
    // ...
}
```

[23] Din acest punct de vedere, funcționalitatea sa este asemănătoare cu a unui obiect de tip sesiune
[24] În acest mod, contextul nu trebuie închis explicit, întrucât acest lucru se produce la sfârşitul blocului `try`-cu-resurse, în caz contrar trebuie apelată metoda `close` pentru a închide conexiunea dacă aplicația şi-a terminat sarcinile. Trebuie să se asigure faptul că toate operațiile din cadrul blocului `try`-cu-resurse sunt executate.
[25] Alternativ, se poate specifica un mesaj de confirmare al mesajelor specific în care clientul transmite în mod explicit confirmare (`JMSContext.CLIENT_ACKNOWLEDGE`), respectiv în care sesiunea transmite întârziat confirmările (`JMSContext.DUPS_ON_ACKNOWLEDGE`).

Un consumator de mesaje permite unui client JMS să îşi exprime interesul vis-a-vis de un furnizor de servicii de mesagerie, acesta fiind responsabil de livrarea mesajelor de la destinație la consumatorii asociați acesteia. Dacă la construirea unui consumator de mesaje se foloseşte un obiect `JMSContext`, livrarea de mesaje începe imediat, un astfel de comportament putând fi dezactivat prin apelul metodei `setAutoStart(false)` (atunci când se creează obiectul `JMSContext`) şi apoi apelând metoda `start` explicit pentru a începe procesul de livrare al mesajelor, respectiv `stop` pentru a-l întrerupe temporar.

Metoda `receive` este utilizată pentru a consuma mesajele în mod sincron, aceasta putând fi apelată la orice moment după construirea unui consumator. Dacă nu se indică nici un argument sau argumentul este 0, metoda se blochează indefinit până când e primit un mesaj (`consumer.receive()` = `consumer.receive(0)`). Metoda poate fi apelată cu un argument ce indică timpul de aşteptare, după care aceasta se va întoarce (cu un rezultat `null`) chiar şi în situația în care mesajul nu a fost primit.

```
Message message = consumer.receive(Constants.TIME_OUT);
```

Dacă se doreşte ca livrarea de mesaje să se desfăşoare asincron dintr-o aplicație client sau dintr-un client Java SE, trebuie folosit **un ascultător de mesaje**, adică un obiect care funcționează ca un proces de gestiune al evenimentelor de primire a mesajelor. Acesta implementează interfața `MessageListener`, care conține o singură metodă, `onMessage`, în care se definesc operațiile ce trebuie realizate la livrarea unui mesaj. Asocierea dintre un consumator de mesaje şi un ascultător de mesaje se face prin metoda `setMessageListener`. Furnizorul JMS apelează în mod automat `onMessage` de fiecare dată când este primit un mesaj. Aceasta primeşte un argument de tip `Message` care poate fi convertit la alt tip de mesaj, în cazul în care este necesar. În cazul containerelor Java EE EJB / web se folosesc componente orientate pe mesaje pentru livrarea asincronă[26]. Metoda onMessage ar trebui să gestioneze orice fel de excepții. Generarea unei excepții de tip `RuntimeException` este considerată eroare de programare.

În situația în care aplicația are nevoie de filtrarea mesajelor pe măsură ce acestea sunt primite, se poate folosi un **selector de mesaje** JMS, care permite consumatorului de mesaje pentru o destinație să indice prin ce se caracterizează care îl interesează. Astfel, sarcina de filtrare a mesajelor este transferată furnizorului de gestiune al mesajelor, în detrimentul aplicației. Un selector de mesaje este un obiect `java.lang.String` care conține o expresie bazată pe un subset al sintaxei SQL92 pentru expresii condiționale. Metoda `createConsumer` şi cele derivate din aceasta permit specificarea unui selector de mesaje ca argument când se construieşte un consumator de mesaje. Astfel, consumatorul de mesaje va primi doar mesajele ale căror antete şi proprietăți se potrivesc cu cele specificate în selectorul de mesaje. Un selector de mesaje nu poate selecta mesajele pe baza conținutului acestora.

```
JMSConsumer consumer =
session.createConsumer(queue,
                       Constants.USER_NAME_PROPERTY + "= '" + userName + "'");
```

[26] O componentă orientată pe mesaje implementează de asemenea interfața `MessageListener` şi conține metoda `onMessage`.

Semantica consumării mesajelor din cadrul unei teme este mai complexă decât în cazul unei cozi. O aplicație consumă mesajele dintr-o temă prin crearea unui abonament în cadrul acesteia, creând un consumator asociat acestuia. Abonamentele pot fi tranzitive sau durabile şi pot fi partajate sau nepartajate. Un abonament poate fi gândit ca o entitate în cadrul furnizorului JMS în timp ce consumatorul este un obiect în cadrul aplicației.

O subscripție va primi o copie a fiecărui mesaj transmis în cadrul temei după ce abonamentul este creat, cu excepția cazului în care a fost selectat un selector de mesaje[27].

Unele abonamente sunt limitate unui singur consumator. În acest caz, toate mesajele din cadrul subscripției sunt livrate către acesta. Alte subscripții permit mai mulți consumatori. Într-o astfel de situație, fiecare mesaj primit de către aceasta va fi livrat către un singur consumator[28].

Subscripțiile pot fi **tranzitive** sau **durabile**.

Un abonament tranzitiv există atâta vreme cât există un consumator activ în cadrul său. Astfel, orice mesaj transmis către temă va fi adăugat la subscripție doar dacă un consumator există şi nu este închis. Un abonament tranzitiv poate fi nepartajat sau partajat.

O subscripție tranzitivă nepartajată nu are un nume şi poate avea asociat doar un singur obiect consumator. Este creată automată automat atunci când se construieşte obiectul consumator. Nu este persistentă, fiind ştearsă automat dacă obiectul consumator este închis. Metoda `JMSContext.createConsumer` creează un consumator de mesaje pe o subscripție tranzitivă nepartajată dacă destinația este o temă.

O subscripție tranzitivă partajată este identificată printr-o denumire şi un identificator al clientului (opțional), putând avea asociate mai multe obiecte de tip consumator. Este creată automat atunci când este construit primul obiect consumator. Nu este persistentă, fiind ştearsă automat dacă şi ultimul obiect consumator este închis.

Deşi implică costuri mai mari, un abonament poate fi durabil, acesta fiind persistent şi continuând să acumuleze mesaje până la momentul în care este şters explicit, chiar dacă nu mai există obiecte de tip consumator care să preia mesaje din cadrul său. O astfel de abordare este necesară atunci când este necesar să se asigure că aplicația va primi toate mesajele transmise.

Ca şi în cazul subscripțiilor tranzitive, un abonament durabil poate fi nepartajat sau partajat.

Un abonament durabil nepartajat este caracterizat printr-o denumire şi un identificator de client (ce trebuie specificat) putând avea un singur consumator asociat cu el.

Un abonament durabil partajat se caracterizează printr-o denumire şi un identificator de client (opțional), putând avea asociate mai multe obiecte de tip consumator de mesaje.

O subscripție durabilă care există dar nu are asociat nici un obiect de tip consumator de mesaje deschis este considerată inactivă.

---

[27] În situația în care a fost specificat un selector de mesaje, doar mesajele ale căror proprietăți întrunesc condițiile specificate vor fi primite de subscripție.

[28] JMS nu defineşte modul în care mesajele sunt distribuite între mai mulți consumatori din cadrul aceluiaşi abonament.

Metoda `JMSContext.createDurableConsumer` poate fi folosită pentru construirea unui consumator de mesaje asociat unui abonament durabil nepartajat. O astfel de subscripție poate avea doar un singur consumator la un moment dat. Un consumator identifică abonamentul durabil din care va primi mesaje prin specificarea unei valori unice reținute de furnizorul JMS. Următoarele obiecte de tip consumator de mesaje ce utilizează acelaşi identificator unic reiau subscripția din starea în care a fost lăsată anterior. Dacă un abonament durabil nu are nici un consumator activ, furnizorul JMS reține mesaje corespunzătoare până când sunt primite de un consumator sau până când expiră. Identitatea unei subscripții durabile poate fi determinată fie prin specificarea unei valori unice[29] pentru întreaga conexiune sau prin indicarea unei teme şi a unei denumiri pentru abonamentul respectiv. Dacă un consumator nu mai este necesar, acesta va fi închis prin apelarea metodei `close`:

```
JMSConsumer consumer =
            context.createDurableConsumer(topic, Constants.SUBSCRIPTION_NAME);
// ...
consumer.close();
```

Furnizorul JMS stochează mesajele transmise către temă în acelaşi mod în care procedează şi în cazul unei cozi. Dacă o altă aplicație va apela metoda `createDurableConsumer` folosind aceeaşi fabrică de conexiuni şi acelaşi identificator pentru client, aceeaşi temă şi aceeaşi denumire a subscripției, atunci subscripția este reactivată şi furnizorul JMS livrează mesajele care au fost transmise pe perioada în care subscripția a fost inactivă.

Pentru a şterge o subscripție durabilă, trebuie închişi toți consumatorii de mesaje, după care trebuie apelată metoda `unsubscribe` primind ca parametru numele său, aceasta ştergând inclusiv starea pe care furnizorul JMS o menține.

```
context.unsubscribe(Constants.SUBSCRIPTION_NAME);
```

O subscripție durabilă partajată permite asocierea mai multor consumatori de mesaje. În acest caz, fabrica de conexiuni utilizată nu trebuie să aibă un identificator al clientului.

Un abonament asociat unei teme creat de metodele `createConsumer` sau `createDurableConsumer` poate avea un singur consumator (deşi o temă poate avea mai multe obiecte de acest tip). Astfel, mai mulți clienți care consumă din cadrul aceleiaşi teme au, prin definiție, mai multe subscripții asociate şi toți clienții primesc toate mesajele transmise către tema respectivă[30].

Totuşi, există posibilitatea de a crea o subscripție tranzitivă partajată pentru o temă folosind metoda `createSharedConsumer` şi specificând nu doar destinația (denumirea temei) ci şi a abonamentului în sine. Prin intermediul unei subscripții partajate, mesajele vor fi distribuite între mai mulți clienți care folosesc aceeaşi temă şi aceeaşi denumire a subscripției. Fiecare mesaj va fi adăugat fiecărei subscripții[30], dar fiecare mesaj de acest tip va fi livrat către un singur consumator din cadrul unei subscripții, deci va fi primit doar de către un singur client, comportamentul fiind util dacă se doreşte partajarea încărcării.

[29] Un astfel de identificator poate fi stabilit administrativ pentru o fabrică de conexiuni specifică unui client.
[30] O excepție de la acest caz este situația în care sunt implementate selectoare de mesaje.

Această funcționalitate îmbunătățeşte scalabilitatea aplicațiilor client, atât pentru Java SE cât mai ales în cazul Java EE[31].

De asemenea, pot fi create abonamente durabile partajate. Spre a construi un astfel de obiect, se apelează metoda `JMSContext.createSharedDurableConsumer` specificând denumirea temei, respectiv a subscripției.

Mesajele transmise unei cozi rămân în cadrul acesteia până când consumatorul de mesaje le primeşte. API-ul JMS implementează un obiect `QueueBrowser` ce permite consultarea mesajelor din coadă şi afişarea valorilor conținute de antetele acestora. În acest scop, se foloseşte metoda `createBrowser` pe un obiect `JMSContext`[32]:

```
QueueBrowser browser = context.createBrowser(queue);
```

Totodată, specificația JMS nu oferă posibilitatea de a consulta o temă, deoarece mesajele sunt şterse de obicei din cadrul acesteia de către furnizor imediat ce sunt adăugate, chiar dacă nu există consumatori de mesaje care să realizeze acest lucru[33].

În **tratarea excepțiilor** trebuie să se aibă în vedere faptul că rădăcina tuturor excepțiilor ce pot fi prinse este `JMSException` în timp ce rădăcina pentru toate excepțiile ce nu pot fi prinse este `JMSRuntimeException`.

Mulți utilizatori recurg la JMS deoarece aplicațiile pe care le dezvoltă nu pot gestiona situații precum mesaje pierdute sau mesaje duplicate, fiind necesar ca fiecare mesaj să fie primit o singură dată funcționalitate oferită de distribuțiile dezvoltate de diferiți producători.

Cel mai sigur mod de a produce un mesaj este folosirea tipului `PERSISTENT`[34] şi transferul său în cadrul unei tranzacții. Tranzacțiile permit ca mai multe mesaje să fie transmise sau să fie primite într-o singură operație atomică[35]. Astfel, **tranzacția** este o unitate de lucru în care pot fi grupate o serie de operații, cum ar fi transmiterea şi primirea de mesaje, astfel că acestea fie reuşesc toate, fie eşuează împreună. Cel mai sigur mod de consumare a mesajelor este în cadrul unei tranzacții, fie dintr-o coadă, fie dintr-o temă din cadrul unei subscripții durabile.

Unele funcționalități permit unei aplicații să-şi îmbunătățească performanțele. De exemplu, pentru mesajele se pot indica **timpi de expirare** după o anumită perioadă de timp, astfel încât consumatorii să nu primească informații care nu mai au actualitate. Totodată, mesajele pot fi transmise asincron, iar confirmarea de primire se poate realiza în mai multe moduri. Alte funcționalități nu sunt legate de siguranță, dar se pot dovedi utile în anumite circumstanțe. Astfel, se pot crea destinații temporare ce durează doar pe parcursul conexiunii în care au fost construire.

---

[31] Componentele orientate pe mesaje împart sarcina de a procesa mesajele din cadrul unei teme pe mai multe fire de execuție.

[32] Metoda createBrowser suportă şi un alt argument, indicând un selector de mesaje.

[33] Cu toate că subscripțiile durabile permit mesajelor să rămână în cadrul unei teme în timp ce consumatorul de mesaje nu este activ, specificația JMS nu defineşte nici o facilitate pentru examinarea acestora.

[34] Mesajele JMS au tipul `PERSISTENT` în mod implicit; astfel de mesaje nu vor fi pierdute în cazul producerii unei erori la nivelul furnizorului JMS.

[35] În cazul platformei Java EE, tranzacțiile permit cuplarea operațiilor de transmitere, respectiv primire a mesajelor cu citiri şi scrieri din şi în baza de date în mod atomic.

Activitate de Laborator

Pentru rezolvarea acestui laborator sunt necesare:

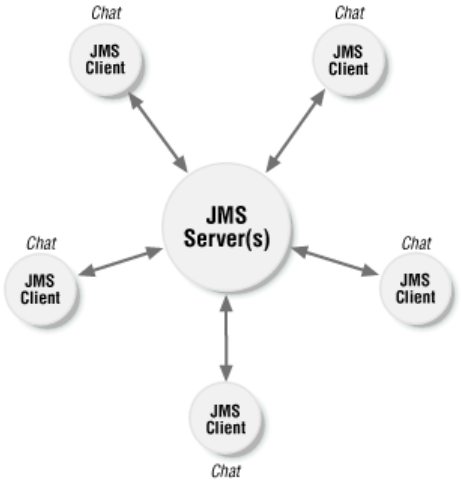
❶ serverul de aplicații GlassFish 4.0

❷ mediul de dezvoltare Eclipse IDE for Java EE Developers 4.3.1 (Kepler)

SAU

❷ mediul dezvoltare NetBeans 7.4

Se doreşte dezvoltarea unui sistem de mesagerie instant utilizând API-ul Java Message Service, format dintr-un server (reprezentat de un obiect `javax.jms.Topic`) şi mai mulți clienți.



Se va lucra în echipe de mai multe persoane. În fiecare echipă pe o maşină va rula serverul (sarcinile specifice fiind marcate prin pictograma), în timp ce pe celelalte maşini vor rula clienți (pictograma marcând sarcinile specifice). Atât pe server cât şi pe client va rula GlassFish 4.0, instanța de pe client redirectând apelurile către instanța de pe server.

Indiferent că rulează pe server (local) sau pe client (la distanță), aplicația JMS va utiliza acelaşi cod sursă, locația obiectelor administrate fiind transparentă pentru acesta.

Serverul este reprezentat de o temă, obiect administrat având denumirea jms/MessagingTopic.

Clienții vor fi pentru aceasta atât producători (transmițând mesaje) cât şi consumatori (primind mesajele care le sunt adresate, fapt asigurat printr-un selector de mesaje, filtrarea făcându-se pe bază de nume de utilizator).

Un mesaj are în antet numele de utilizator al clientului căruia i se adresează, corpul fiind un obiect de tip CustomMessage în care se reține numele de utilizator al clientului care l-a trimis precum şi conținutul.

STRUCTURĂ MESAJ	
utilizat în filtrul de mesaje	Antet `Constants. CUSTOM_MESSAGE_PROPERTY = receiverUserName`

`CustomMessage`
⇧
`ObjectMessage`
	Conținut
	`String senderUserName`
	`String messageContent`

[0p] 1. Să se instaleze serverul de aplicații GlassFish 4.0.

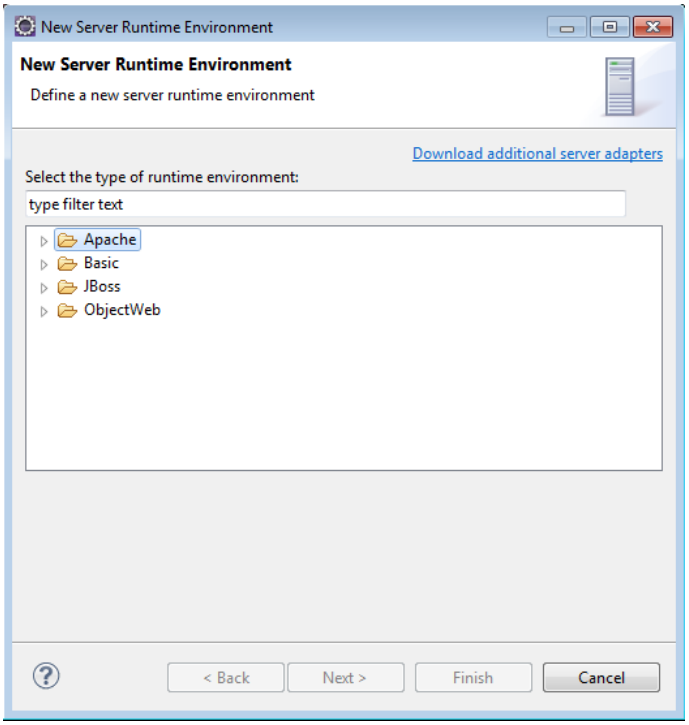
În urma procesului, se vor afișa parametrii cu care a fost configurat serverul de aplicații, constând în parola pentru domeniu (implicit vidă) precum şi porturile specifice diferitelor protocoale de comunicații.

```
Performing the required configurations
_____________________________________

Creating domain
_______________
Executing command :C:\glassfish4\glassfish\bin\asadmin.bat --user admin --
passwordfile - create-domain --
template=C:\glassfish4/glassfish/common/templates/gf/appserver-domain.jar --
savelogin --checkports=false --adminport 4848 --instanceport 8080 --
domainproperties=jms.port=7676:domain.jmxPort=8686:orb.listener.port=3700:http.ssl.
port=8181:orb.ssl.port=3820:orb.mutualauth.port=3920 domain1
C:\glassfish4\glassfish\bin\asadmin.bat --user admin --passwordfile - create-domain
--template=C:\glassfish4/glassfish/common/templates/gf/appserver-domain.jar --
savelogin --checkports=false --adminport 4848 --instanceport 8080 --
domainproperties=jms.port=7676:domain.jmxPort=8686:orb.listener.port=3700:http.ssl.
port=8181:orb.ssl.port=3820:orb.mutualauth.port=3920 domain1 Using port 4848 for
Admin.
Using port 8080 for HTTP Instance.
Using port 7676 for JMS.
Using port 3700 for IIOP.
Using port 8181 for HTTP_SSL.
Using port 3820 for IIOP_SSL.
Using port 3920 for IIOP_MUTUALAUTH.
Using port 8686 for JMX_ADMIN.
Using default port 6666 for OSGI_SHELL.
Using default port 9009 for JAVA_DEBUGGER.
Distinguished Name of the self-signed X.509 Server Certificate is:
[CN=Andrei-PC,OU=GlassFish,O=Oracle Corporation,L=Santa Clara,ST=California,C=US]
Distinguished Name of the self-signed X.509 Server Certificate is:
[CN=Andrei-PC-instance,OU=GlassFish,O=Oracle Corporation,L=Santa
Clara,ST=California,C=US]
Domain domain1 created.
Domain domain1 admin port is 4848.
Domain domain1 allows admin login as user "admin" with no password.
Login information relevant to admin user name [admin]
for this domain [domain1] stored at
[C:\Users\Andrei\.gfclient\pass] successfully.
Make sure that this file remains protected.
Information stored in this file will be used by
administration commands to manage this domain.
Command create-domain executed successfully.


Starting domain
_______________
Executing command :C:\glassfish4\glassfish\bin\asadmin.bat start-domain domain1

C:\glassfish4\glassfish\bin\asadmin.bat start-domain domain1
Attempting to start domain1.... Please look at the server log for more details.....
```

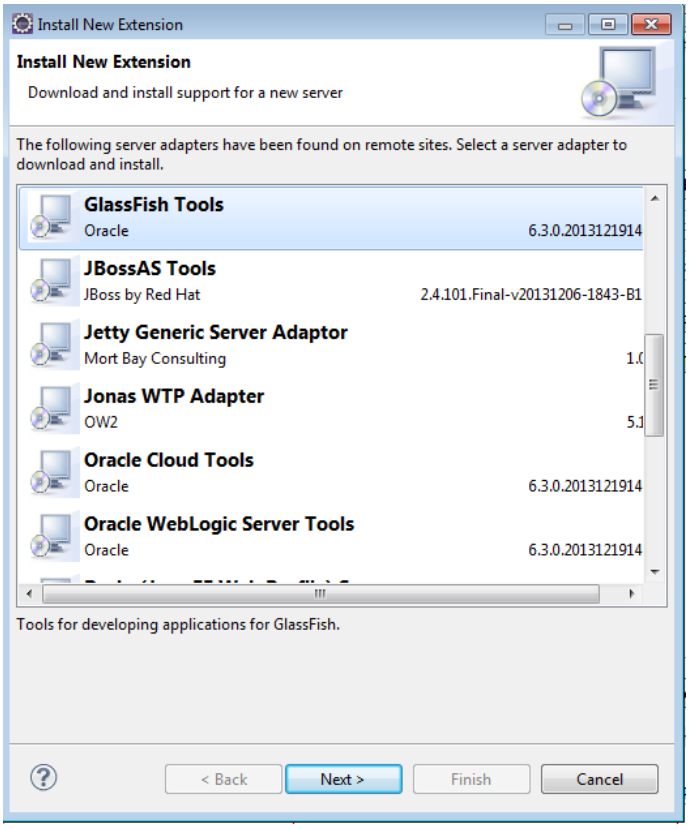
[0p] 2. Să se integreze mediul de dezvoltare Eclipse IDE for Java EE Developers 4.3.1 (Kepler), respectiv mediul de dezvoltare NetBeans 7.4 cu serverul de aplicații GlassFish 4.0.

Acest pas nu este absolut necesar pentru rularea aplicației, însă configurarea mediilor de dezvoltare este utilă întrucât în acest mod pot fi verificate proprietățile serverului de aplicații precum şi starea acestuia în fiecare moment al execuției programului care utilizează specificația JMS.

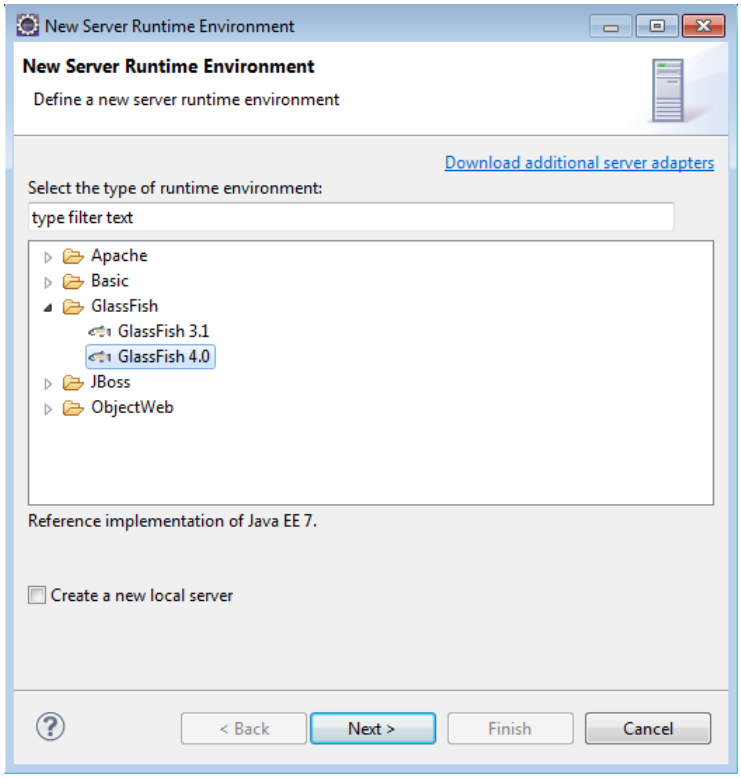
Integrarea mediului de dezvoltare Eclipse IDE for Java EE Developers 4.3.1 (Kepler) cu serverul de aplicații GlassFish 4.0 se face accesând *Window* → *Preferences* → *Server Runtime* → *Environment* → *Add...*.

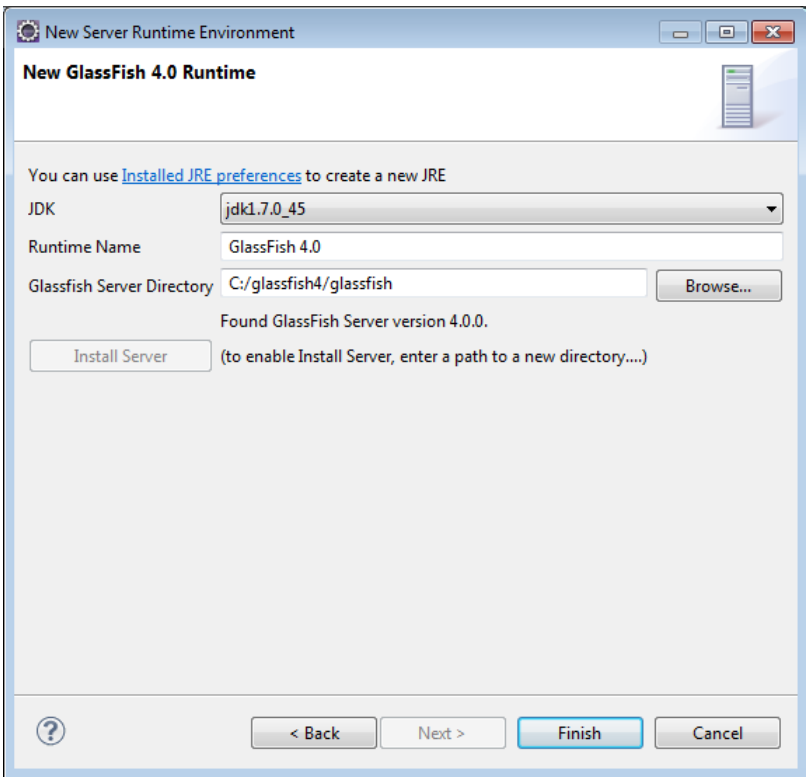


În cazul în care GlassFish nu este listat ca optiune de server de aplicații care să poată fi configurat, se alege *Download aditional server adapters*. Ulterior, se selectează *GlassFish Tools (Oracle)*. După instalarea acestui adaptor, se reporneşte mediul de dezvoltare Eclipse Kepler 4.3.1 for Java EE Developers.

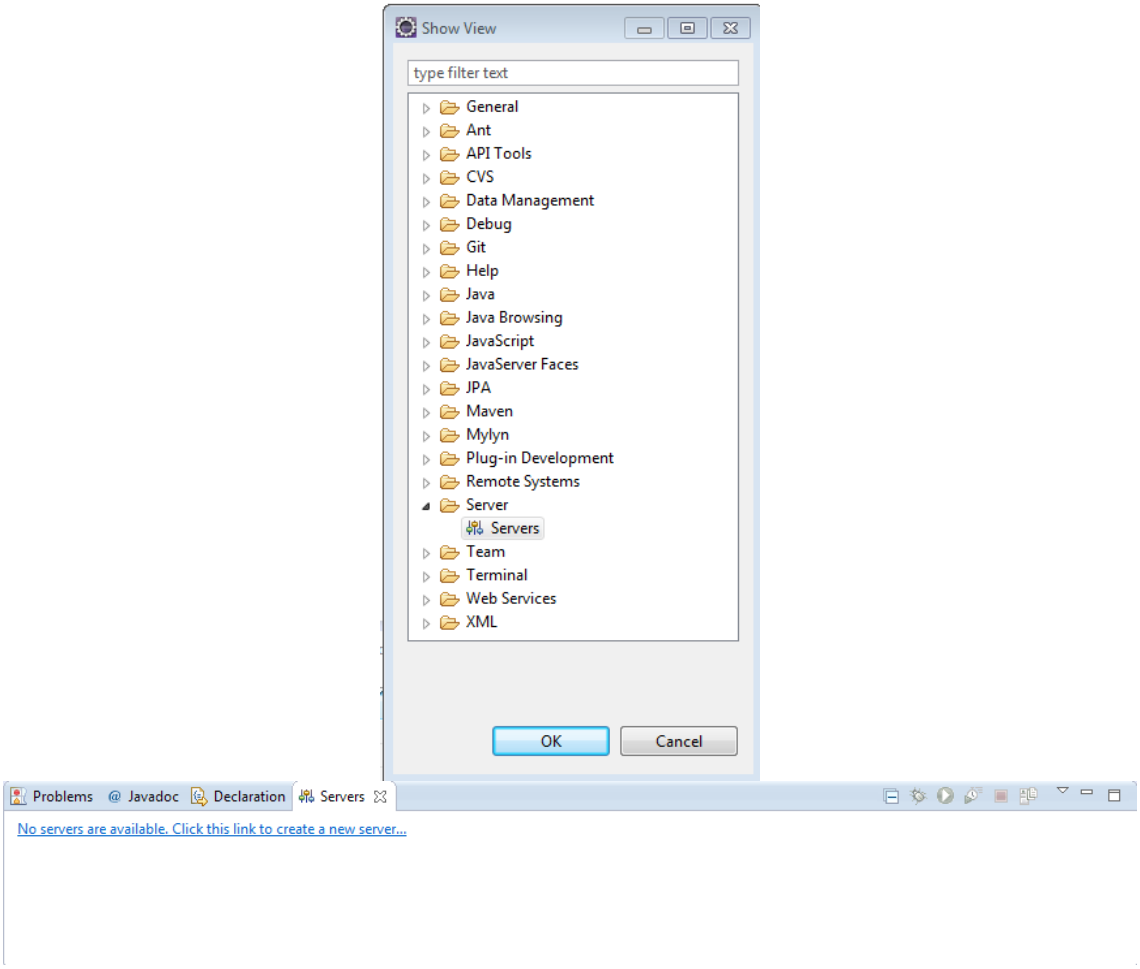


Se selectează serverul de aplicații GlassFish 4.0 precizându-se directorul în care acesta este instalat. În această etapă de configurare este absolut necesar să se precizeze calea către JDK şi nu către JRE (crearea unui astfel de obiect se face din *Window* → *Preferences* → *Java* → *Installed JREs*).

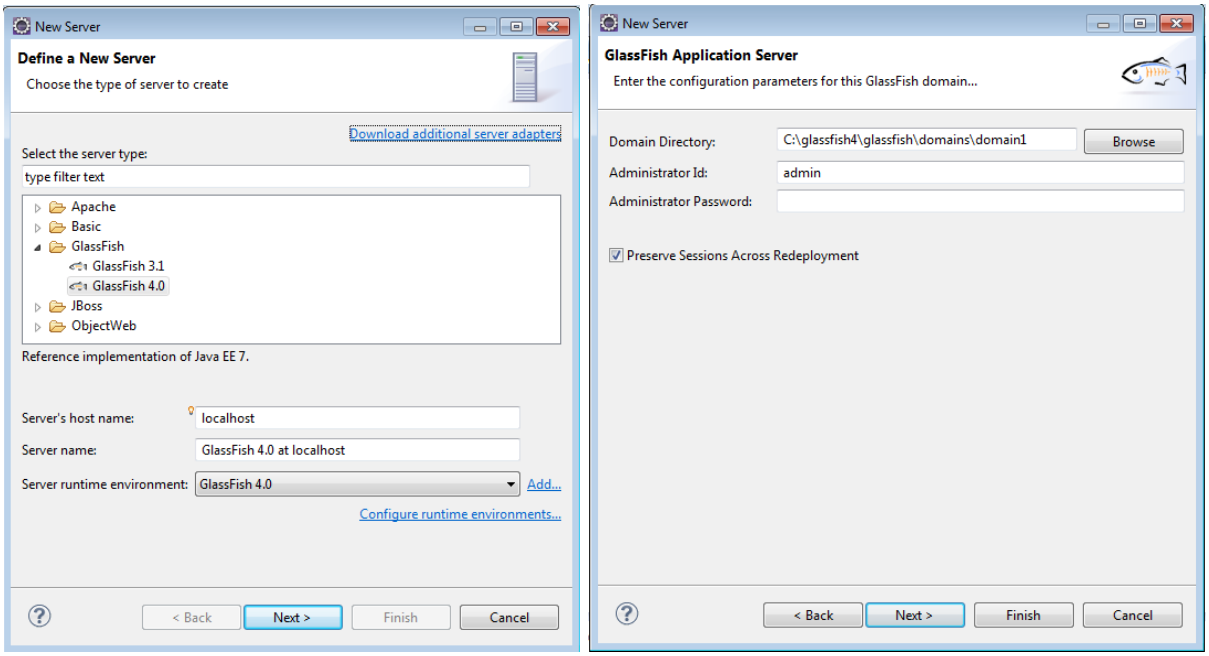




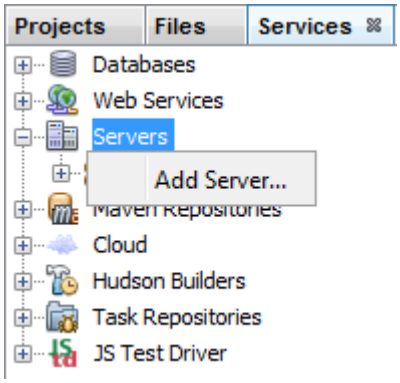
Se deschide perspectiva asociată serverului de aplicații din *Window* → *Show View* → *Other...* → *Server* → *Servers*.



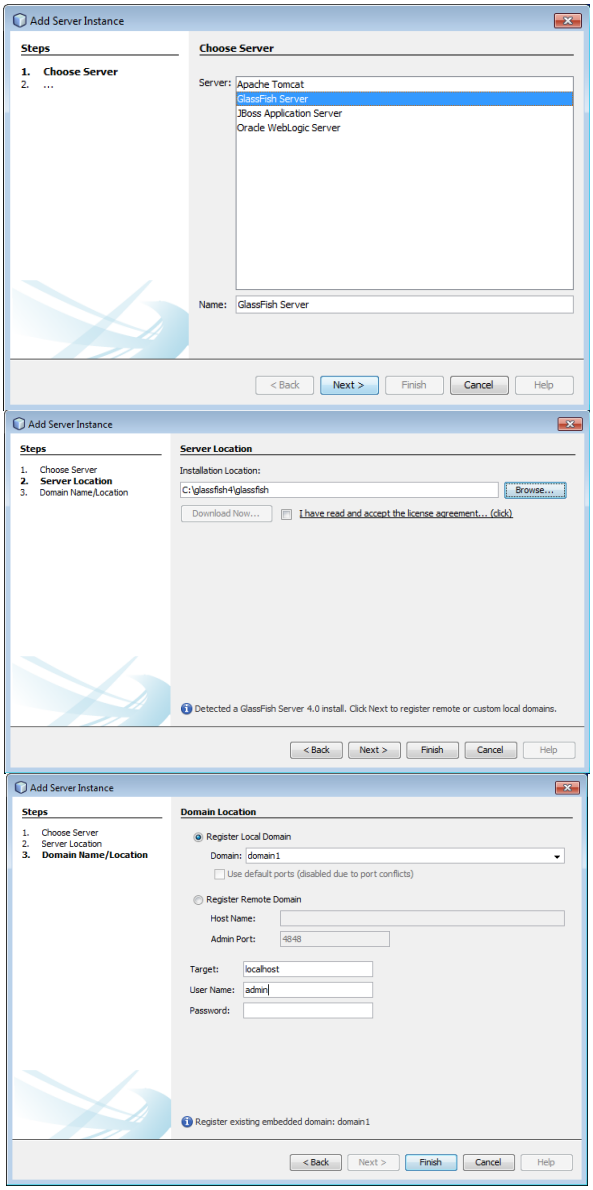
Se integrează serverul de aplicații GlassFish 4.0 în această perspectivă accesând opțiunea *Click this link to create a new server...* şi indicând locația specifică domeniului în care se va lucra, numele de utilizator (admin) şi parola (vidă).



Integrarea mediului de dezvoltare NetBeans 7.4 cu serverul de aplicații GlassFish 4.0 se face accesând *Services* → *Server* → *Add Server*....



Se indică tipul de server (GlassFish Server), locația la care acesta a fost instalat, precum şi câteva informații de configurare (denumirea domeniului local, maşina unde rulează şi informațiile de autentificare – nume utilizator / parola).

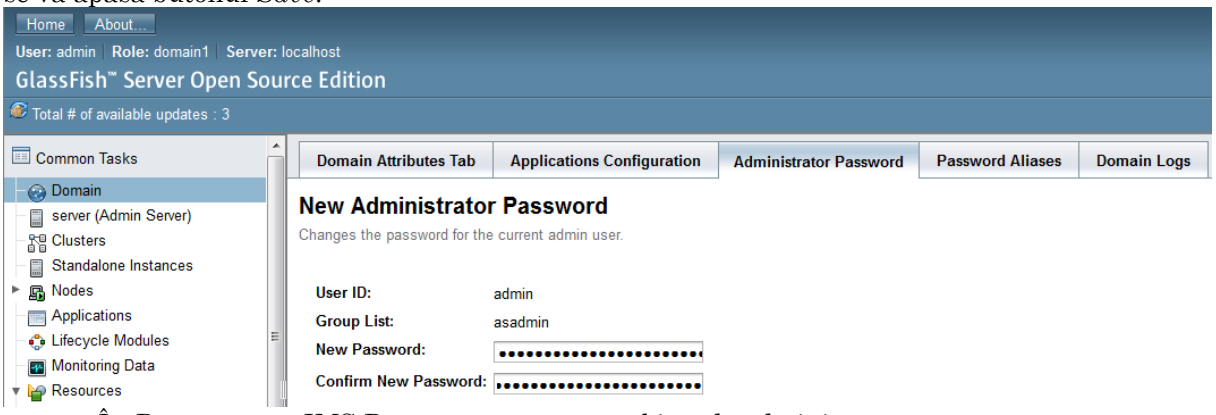


[0,50p] 3. Să se pornească serverul de aplicații GlassFish 4.0.

```
GLASSFISH_HOME/bin>asadmin start-domain domain1
```

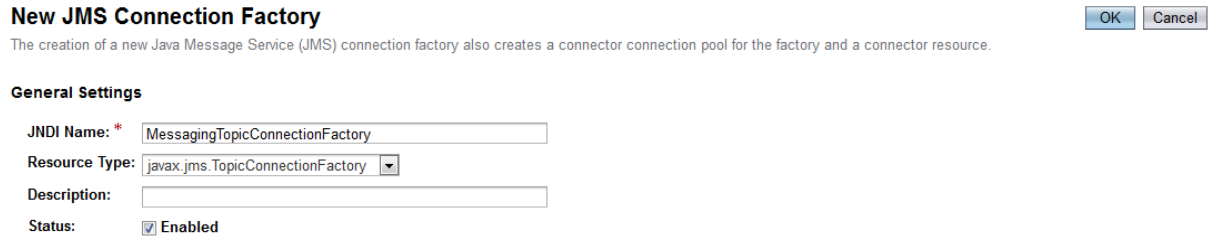
[2p] 4. Să se acceseze consola de administrare a serverului de aplicații GlassFish accesibilă la http://localhost:4848.

În *Domain* → *Administrator Password*, se va specifica o parolă pentru administrator şi se va apăsa butonul *Save*.

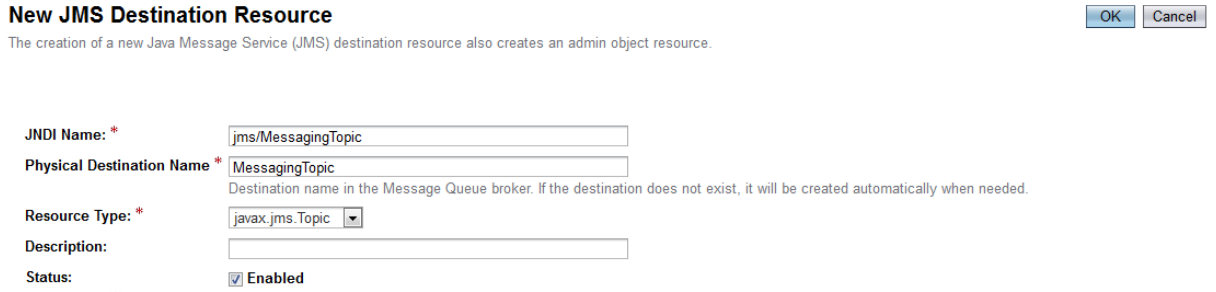


În *Resources* → *JMS Resources* se vor crea obiectele administrate:

a) *Connection Factories* → *New*: fabrica de conexiuni **MessagingTopicConnectionFactory** cu tipul `javax.jms.TopicConnectionFactory` (se apasă butonul *OK*).

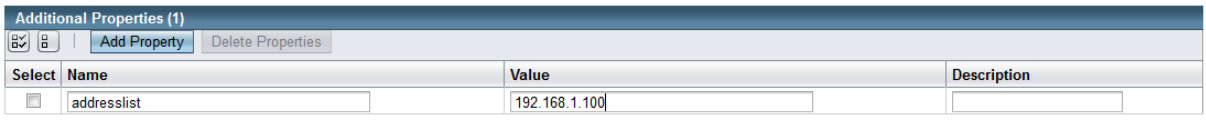


b) *Destination Resources* → *New*: resursa destinație de tip temă **jms/MessagingTopic** având tipul `javax.jms.Topic` (se apasă butonul *OK*).



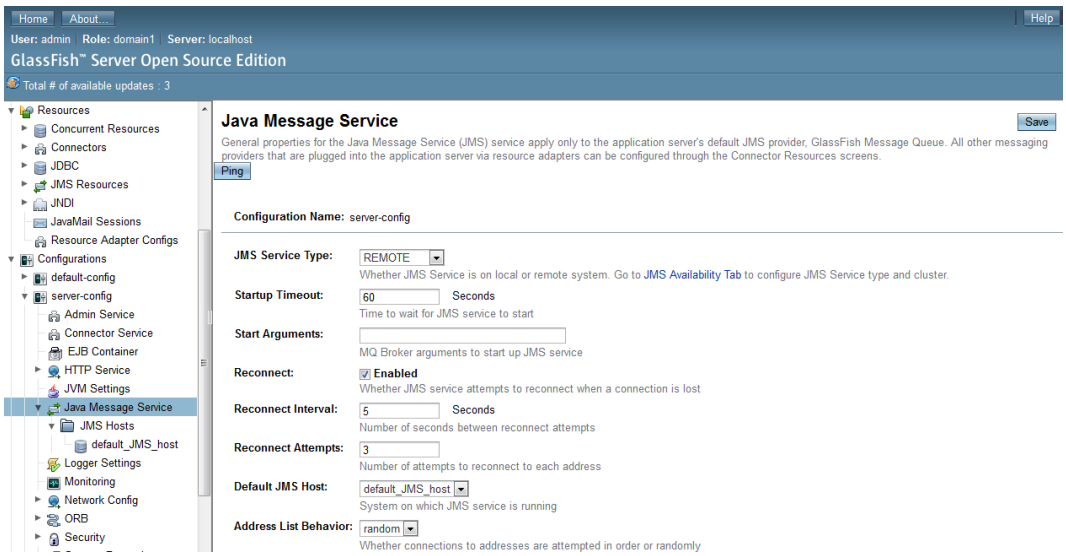
Configurațiile privind **obiectele administrate** (MessagingTopicConnectionFactory şi jms/MessagingTopic) trebuie realizate întocmai şi pe client, cu precizarea faptului că pentru fiecare dintre acestea se vor specifica nişte proprietăți suplimentare (*Additional Properties*), care să redirecteze utilizarea acestor obiecte administrare către corespondentele lor de pe server, motivul definirii lor fiind doar acela al identificării unor referințe (acestea putând fi accesate local din aplicație, nefiind permisă conectarea la distanță în mod direct).

Se va specifica proprietatea **addresslist**, având ca valoare adresa IP a maşinii pe care rulează serverul.



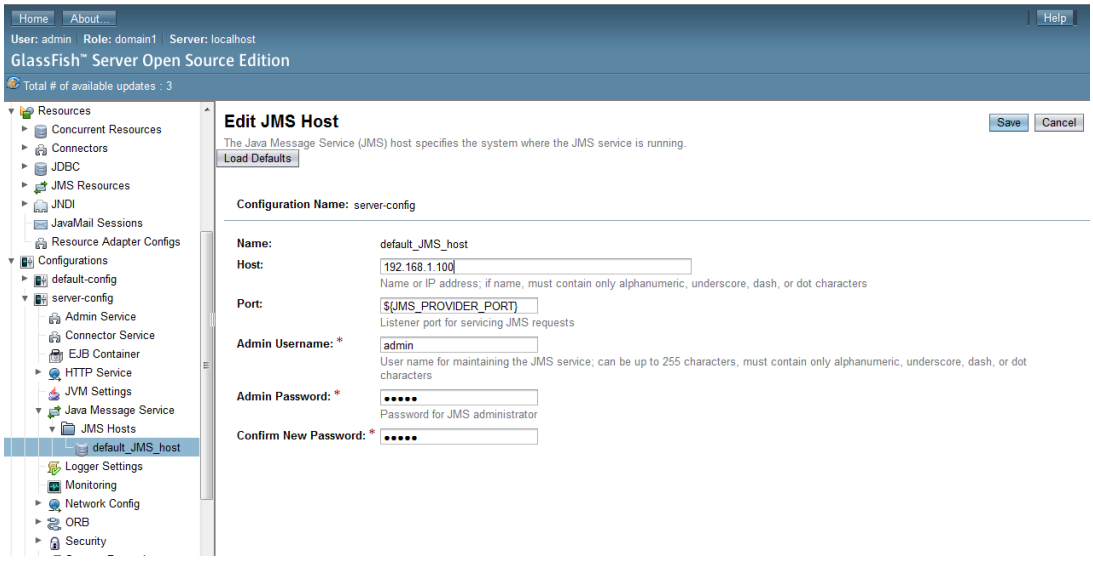
În *Configurations → server-config → Java Message Service*:

a) se specifică **tipul de serviciu JMS** pentru a putea fi accesat la distanță (proprietatea **JMS Service Type** va avea valoarea REMOTE);



Se accesează butonul *Save*.

b) se indică locația la care se află resursele JMS care vor fi accesate (pentru **default_JMS_host** se va specifica câmpul Host, reprezentând adresa IP a serverului GlassFish); serverul şi clientul trebuie să se găsească în aceeaşi (sub)rețea sau trebuie să aibă adrese IP publice, vizibile în Internet; de asemenea, se va indica parola serverului de aplicații GlassFish 4.0 care rulează la locația indicată.



Se accesează butonul *Save*.

[0,50p] 5. Să se adauge bibliotecile lipsă proiectelor Eclipse IDE for Java EE Developers 4.3.1 (Kepler) şi NetBeans 7.4 astfel încât acestea să poată rula:

Messaging Client → Build Path → Configure Build Path → Libraries → Add External Jars
GLASSFISH_HOME/lib/gf-client.jar (dependențele către alte referințe vor fi adăugate automat)

Messaging Client → Libraries → Add Jar/Folder...
GLASSFISH_HOME/lib/*.jar (4 fişiere)
GLASSFISH_HOME/lib/install/applications/jmsra/*.jar (274 fişiere)
GLASSFISH_HOME/lib/modules/*.jar (6 fişiere)

[1p] 6. Să se ruleze aplicația, în care numele de utilizator va fi prenume.nume, transmițându-se mesaje către colegii din cadrul grupei.

[5p] 7. La intrarea, respectiv la ieşirea din aplicație, să se transmită mesajele `Constants.LOGIN_MESSAGE`, respectiv `Constants.LOGOUT_MESSAGE` către toți utilizatorii cu care s-a comunicat recent pentru a se anunța faptul că utilizatorul este activ, respectiv inactiv.

Un utilizator care a primit un mesaj că un alt utilizator este disponibil îl va include în lista utilizatorilor activi.

Un utilizator ce a primit un mesaj că un alt utilizator nu mai e disponibil îl va exclude din lista utilizatorilor activi.

a) În clasa `ContactsList`, în metoda `start`, să se transmită mesajul `Constants.LOGIN_MESSAGE` către toți utilizatorii cu care s-a comunicat recent (din `recentContactsListItem`). Se va apela metoda `publish` a obiectului `communicator` având ca parametrii numele de utilizator al destinatarului şi mesajul propriu-zis.
b) În clasa `ContactsList`, în metoda `handleMessage`, să se analizeze mesajul `Constants.LOGIN_MESSAGE`, parcurgându-se lista `connectedContactsListItem` şi adăugându-se la aceasta numele de utilizator din mesajul primit, în cazul în care utilizatorul nu se regăseşte în aceasta.
c) În clasa `ContactsList`, în metoda `close`, să se transmită mesajul `Constants.LOGOUT_MESSAGE` către toți utilizatorii cu care s-a comunicat recent (din `recentContactsListItem`). Se va apela metoda `publish` a obiectului `communicator` având ca parametrii numele de utilizator al destinatarului şi mesajul propriu-zis.
d) În clasa `ContactsList`, în metoda `handleMessage`, să se analizeze mesajul `Constants.LOGOUT_MESSAGE`, parcurgându-se lista `connectedContactsListItem` şi adăugându-se la aceasta numele de utilizator din mesajul primit, în cazul în care utilizatorul nu se regăseşte în aceasta.

[1p] 8. Să se modifice comportamentul aplicației astfel încât utilizatorii să poată primi şi mesajele transmise în răstimpul în care nu au fost disponibili.

În clasa `PublishSubscribe`, în metoda `subscribe`, se va modifica tipul obiectului consumator de mesaje astfel încât acesta să fie durabil (persistent şi atunci când este inactiv, adică nu are nici un abonat conectat la el).

[1p] 9. Să se modifice formatul mesajelor astfel încât acesta să includă şi ora la care mesajul a fost transmis:

```
utilizator1(zz1/ll1/aaaa1 hh1:mm1)> mesaj1
utilizator2(zz2/ll2/aaaa2 hh2:mm2)> mesaj2
```

Se va modifica structura mesajului din clasa `CustomMessage`, modificându-se definiția metodei `handleMessage` din clasa `ContactsList` astfel încât să conțină şi parametrul referitor la momentul de timp la care a fost transmis mesajul.

Bibliografie

Eric JENDROCK, Ricardo CERVERA-NAVARRO, Ian EVANS, Devika GOLLAPUDI, Kim HAASE, William MARKITO, Chinmayee SRIVATHSA, *The Java EE 7 Tutorial, Release 7 for Java EE Platform*, 2013