Assicurarsi che la transazione sia finita

Come eseguire un'operazione avendo la certezza che una transazione sia terminata?

In questo articolo cercheremo di capire che cosa si intende per transazione e come intercettare il completamento di una transazione all'interno di un metodo dei services

Cenni sul concetto di transazione

Quando parliamo di "transazione" in ambito informatico intendiamo una serie di operazioni, tipicamente che coinvolgono scritture su database, eseguite in sequenza. Eseguire queste operazioni in una transazione significa che, se anche una sola di queste fallisce, allora vengono annullate anche tutte le altre precedentemente eseguite e la transazione si interrompe: in altre parole viene eseguito un rollback della transazione

Una caratteristica importante di una transazione è la coerenza dei dati al suo interno: anche se la transazione non è completata e i dati quindi non ancora persistiti su database, eventuali query eseguite all'interno della transazione su questi dati ritornano risultati coerenti con le operazioni compiute.

Ad esempio immaginiamo un'operazione di trasferimento di denaro dal conto A al conto B tramite un sistema online: possiamo identificare 2 macro operazioni che vanno a modificare i dati:

1. Prelievo dal conto A

2. Aggiunta sul conto B

Immaginiamo che si verifichi un errore nell'esecuzione della fase (2): la transazionalità di questo flusso ci assicura che l'intera operazione verrà annullata e il conto A rimarrà immutato. Se invece le operazioni non fossero state eseguite in modo transazionale la cifra prelevata da A sarebbe andata persa.
Inoltre la transazionalità ci assicura che eseguendo una verifica del saldo del conto A tra (1) e (2), la cifra ritornata sarà coerente con il prelievo effettuato, nonostante ancora l'operazione non sia stata effettivamente eseguita sul database.

All'interno di Liferay la transazionalità è garantita per tutte le operazioni eseguite all'interno dei service che coinvolgono entità su database: il rollback scatta quando vengono lanciate eccezioni di tipo SystemExcetion e PortalException.

Possibili problemi di coerenza di dati tra diverse transazioni

Ci possono essere casi in cui dobbiamo compiere delle operazioni in una transazione differente da quella che sta modificando i dati.

Ad esempio se implementiamo un Model Listener e nel metodo afterCreate cerchiamo di fare una findEntity dell'entità appena creata, riceveremo null invece dell'entità. Questo perché il model listener viene eseguito in una transazione separata da quella che ha generato l'entità, che potrebbe non essere ancora completata.

Oppure se usiamo il message bus per informare plugin esterni della creazione/modifica di un'entità (vedi http://marcorosce.cluster020.hosting.ovh.net/comunicazio

ne-tra-portlet-con-messagebus/) può succedere che il listener scatti prima che la transazione originaria sia completata, creando disallineamenti e problemi.

Intercettare il completamento di una transazione

Per intercettare il completamento di una transazione possiamo usare il metodo registerCallback della classe TransactionCommitCallbackRegistryUtil.

Questo metodo ci permette di registrare un task (che implementi l'interfaccia Callable) che verrà eseguito solamente al completamento della transazione.

Approfondimenti e riferimenti

- https://en.wikipedia.org/wiki/Database_transaction
- http://www.liferayaddicts.net/blogs/-/blogs/transaction-management-with-liferay-service
- http://proliferay.com/liferay-service-builder-transaction/

Aggiungere icon font personalizzati

Cosa sono le icon font? A cosa

servono?

Nello sviluppo di applicazioni ci troviamo spesso a dover inserire icone all'interno delle nostre interfacce utente e solitamente utilizziamo immagini bitmap (bmp, jpeg, gif, png, etc.) recuperate in rete o create ad hoc. Questo approccio però porta con se una serie di inconvenienti come ad esempio problemi di scalabilità con risoluzioni differenti (ad esempio su interfacce responsive), aumento del consumo di banda e una certa difficoltà in caso decidessimo, ad esempio, di cambiare l'aspetto dell'immagine in base ad una selezione.

Per risolvere questi problemi possiamo utilizzare le icon immagine e, come tali, godono di tutti vantaggi dei font tra i quali:

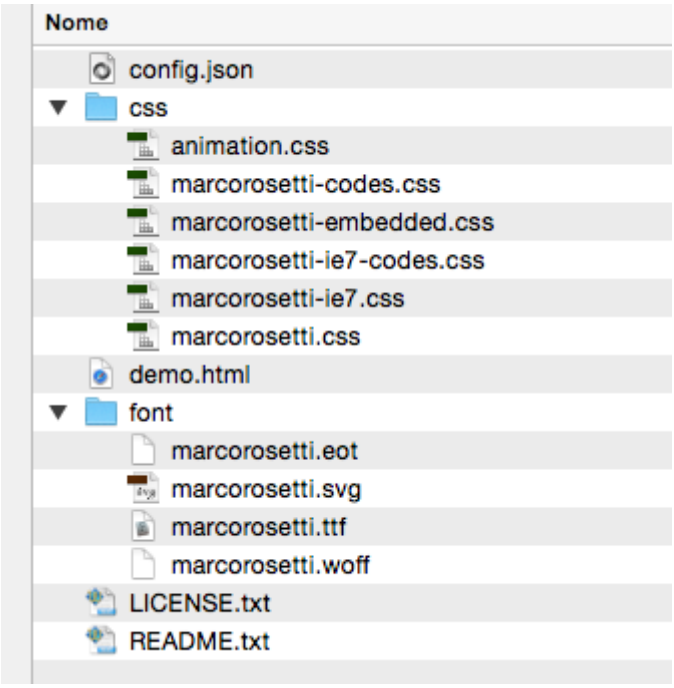
- Sono vettoriali e quindi scalano senza sgranare
- Possono essere cambiati di colore tramite css
- Possono essere ombreggiati tramite css
- Occupano meno spazio (e quindi utilizzano meno banda) rispetto ad un'immagine

A partire dalla versione 6.2 Liferay include al suo interno un pacchetto di icon font (la lista è disponibile qui: http://liferay.github.io/alloy-bootstrap/base-css.html#icons) e possono essere utilizzati out-of-the-box.

Esistono in rete molti siti che permettono di creare, anche in modo gratuito, dei set di icon font scegliendo da elenchi interminabili di possibilità quelle che ci interessano. Citiamo ad esempio http://www.fontello.com/, http://www.flaticon.com/ e https://icomoon.io/. In questo articolo vedremo come utilizzare le icon font di un pacchetto custom all'interno della nostra portlet.

Creare un pacchetto personalizzato di icon font

Prendiamo come esempio il sito Fontello (http://www.fontello.com/). Possiamo selezionare tutte le icone che vogliamo inserire all'interno del nostro set e scaricarle in un unico archivio zip.



Vediamo nel dettaglio cosa contiene l'archivio creato:

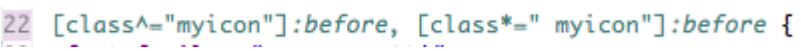
- Cartella CSS – i fogli di stile che tra poco copieremo all'interno del nostro plugin
- Cartella Font – i file contenenti il font vero e proprio, anche questi dovranno essere copiati nel nostro plugin
- config.json – un file in formato JSON che contiene le indicazioni su tutte le icone che fanno parte del nostro pacchetto. Caricando questo file sul sito avremo tutte le nostre icone già selezionate e impostate: in questo modo è più semplice modificare il nostro set senza dover tutte le volte ripartire da capo
- demo.html – file di demo dell'intero set di icone da aprire in qualsiasi browser

- LICENSE.txt – file di licenza
- README.txt – istruzioni di installazione. Da leggere sempre!!

Aggiungere le icon font al proprio plugin

1. Copiare i file dei font all'interno di docroot. Io consiglio di utilizzare la sottocartella fonts ma non è obbligatorio
2. Copiare i file css all'interno di docroot. Anche in questo caso nessun path è obbligatorio ma consiglio di utilizzare la sottocartella css. In realtà per il nostro esempio ci interessa solo il file senza suffisso, nel nostro caso marcorosetti.css
3. Apriamo il file marcorosetti.css e per prima verifichiamo (ed eventualmente correggiamo) i path relativi ai font che troviamo all'interno dell'istruzione @font-face

4. Assicuriamoci che il selettore css e le classi del set non inizino con "icon-" o potremmo ritrovarci con alcune incompatibilità con gli icon font di default di liferay. Possiamo modificare il file css a mano oppure utilizzare le impostazioni avanzate di esportazione di fontello. Il risultato da ottenere dovrà essere simile a questo:



```
56
57 .myiconsun:before { content: '\e800'; }
58 .myiconcloud-sun:before { content: '\e801'; }
59 .myiconfog-sun:before { content: '\e802'; }
60 .myiconcloud-flash-alt:before { content: '\e803'; }
61 .myiconsnow:before { content: '\e804'; }
62 .myiconrain:before { content: '\e805'; }
```

5. Sempre per questioni di compatibilità meglio rimuovere le direttive su width e margin-right:

```
23  font-family: "marcorosetti"
24  font-style: normal;
25  font-weight: normal;
26  speak: none;
27
28  display: inline-block;
29  text-decoration: inherit;
30  /*width: 1em;
31  margin-right: .2em;*/
32  text-align: center;
33  /* opacity: .8; */
34
```

6. Infine aggiungiamo al file liferay-portlet.xml i riferimenti ai css <header-portlet-css>/css/marcorosetti.css</header-portlet-css> in modo che il css venga caricato automaticamente su ogni nostra pagina

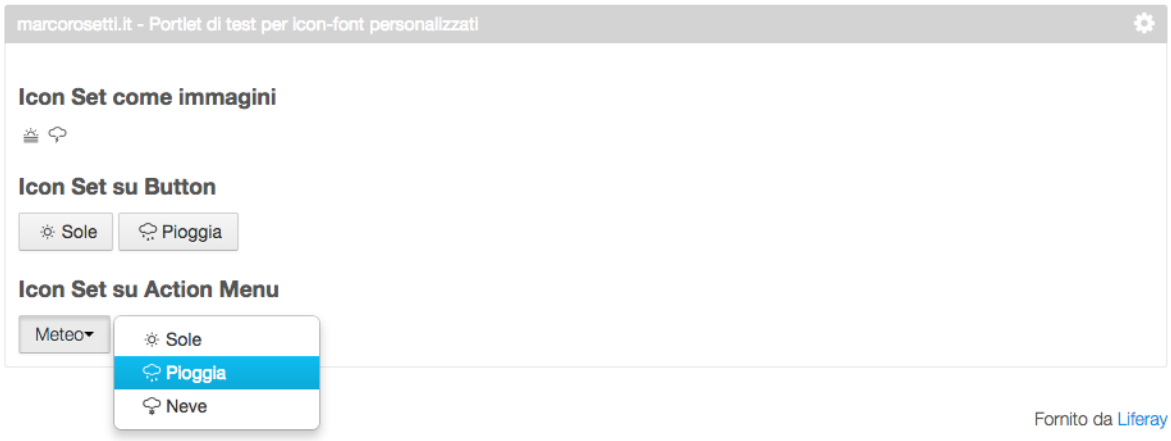
```
15      <portlet>
16          <portlet-name>Icon Font Test</portlet-name>
17          <icon>/icon.png</icon>
18          <header-portlet-css>/css/main.css</header-portlet-css>
19          <header-portlet-css>/css/marcorosetti.css</header-portlet-css>
20          <footer-portlet-javascript>
21              /js/main.js
22          </footer-portlet-javascript>
23          <css-class-wrapper>icon font test-portlet</css-class-wrapper>
24      </portlet>
```

Esempi di utilizzo

A seconda del componente in cui vogliamo inserire l'icona l'utilizzo è leggermente diverso.

Ecco il codice di una piccola jsp di esempio per l'utilizzo come semplice immagine (tag <i>), in un pulsante (tag <aui:button>) e tra le azioni di un icon-menu (tag <liferay-ui:icon> all'interno di <liferay-ui:icon-menu>)

E questo è il risultato finale



Approfondimenti e riferimenti

- http://www.html.it/articoli/icon-fonts-1/
- http://www.fontello.com/
- http://www.flaticon.com/
- http://www.vanseodesign.com/web-design/icon-fonts/
- https://css-tricks.com/examples/IconFont/

Comunicazione tra plugin grazie al Message Bus

Come far comunicare tra loro due plugin differenti?

Ci sono vari modi per trasferire informazioni tra due plugin differenti. Uno di questi è lo scambio di messaggi tramite Message Bus.

Possiamo individuare due ruoli principali in questa comunicazione:

- *Message producer* – Il plugin che crea il messaggio
- *Message consumer* – Il plugin che riceve il messaggio

Vediamo quali sono i passi necessari per implementare questo tipo di comunicazione

Message producer – Creare la Destination

Per prima cosa occorre creare una destinazione per i nostri messaggi. Questa operazione può essere fatta nel file messaging-spring.xml ma in questo modo ogni volta che il plugin sarà deployato la destination verrà ricreata, perdendo tutti i listener eventualmente associati. Un modo per ovviare a questo problema è creare la destination tramite codice java:

Il miglior momento in cui invocare questo metodo è all'interno del metodo di startup del plugin. Per i dettagli vedere QUI

Message producer – Inviare il messaggio

Per inviare il messaggio possiamo utilizzare in qualsiasi punto del nostro codice alcuni metodi statici messi a disposizione da Liferay. Il messaggio creato può

Message consumer – Implementare il listener

Per prima cosa dobbiamo implementare la classe che gestirà la ricezione del messaggio. Per farlo dobbiamo estendere la classe com.liferay.portal.kernel.messaging.MessageListener: in particolare la logica di gestione del messaggio deve essere implementata nel metodo void receive(Message message):

Message consumer – Agganciare il listener alla destination

Una volta implementato il listener va agganciato alla destinazione creata. Ricordiamoci di gestire anche il redeploy del plugin, quindi di controllare se è già presente un listener del tipo implementato prima di aggiungerlo. Per farlo possiamo usare codice di questo tipo:

Non facciamoci spaventare da un codice all'apparenza più complesso del solito: si tratta di scorrere tutta la lista dei listener agganciati alla destinazione ed eventualmente sganciare il listener della classe MyListener. La complicazione nasce dal fatto che in alcuni casi viene utilizzato il proxy InvokerMessageListener per wrappare la vera classe del listener. Non ci addentriamo nei dettagli di questa scelta ma ci basta sapere come gestirla: invocando il metodo getMessageListener() sul proxy, che ci restituisce la vera istanza del listener e ci permette di verificarne la classe.

Anche in questo caso conviene fare questa operazione all'interno del codice di startup del plugin in modo da essere sicuri che venga eseguita sempre quando il plugin è presente

Approfondimenti e riferimenti

- https://www.liferay.com/it/documentation/liferay-portal/6.1/development/-/ai/lp-6-1-dgen09-using-message-bus-0

Indicizzazione entità custom con Lucene

Come possiamo utilizzare Lucene per la ricerca delle nostre entità custom?

Liferay utilizza il motore di indicizzazione Lucene per eseguire le ricerche su molte delle sue entità di portale. In questo articolo vedremo quali sono i passi per implementare ricerche di questo tipo anche sulle entità delle nostre portlet custom.

Implementare l'indexer

La prima cosa da fare è descrivere come tradurre la nostra entità custom in modo che Lucene possa gestirla: per fare questo dobbiamo implementare un nostro "indexer". L'indexer sarà una classe che deve estendere la classe astratta com.liferay.portal.kernel.search.BaseIndexer

I metodi principali che dobbiamo implementare:

- String[] getClassNames() – fornisce un array dei nomi dei modelli che il nostro indexer gestisce
- String getPortletId() – fornisce l'identificativo univoco della nostra portlet
- String getPortletId(SearchContext searchContext) – analogo al metodo precedente
- void doDelete(Object entity) – le operazioni da eseguire sugli indici Lucene quando viene eliminata un'entità (ad es. eliminare anche le entità correlate)
- Document doGetDocument(Object entity) – il metodo principale, in cui dobbiamo implementare la logica di mapping tra la nostra entità custom e l'oggetto di tipo Document gestito direttamente da Lucene e salvato sugli

indici

- Summary doGetSummary(Document document, Locale locale, String snippet, PortletURL portletUrl) – restituisce una rappresentazione sommaria di un oggetto Document proveniente da una ricerca su Lucene. Sebbene questo metodo non sia strettamente necessario per l'utilizzo degli indici di lucene, viene utilizzato in molte portlet del portale, quali Asset Publisher o Ricerca, quindi consiglio di implementarlo almeno in bozza
- void doReindex(Object object) – invocato dal portale tutte le volte che è necessario il reindex. Il parametro in ingresso generico permette di gestire diversi casi, ad esempio:
    - List<[MyEntity]> – una lista di entità da indicizzare
    - [MyEntity] – un'unica entità da indicizzare
    - long[] – un array di chiavi primarie delle entità da indicizzare
    - etc.
- void doReindex(String[] companyIds) – invocato quando si effettua il reindex da pannello di controllo (Vedi "Reindex completo entità della portlet" per dettagli). In ingresso vengono ricevuti gli id di tutte le company presenti nel portale
- void doReindex(String className, String classPK) – invocato quando è necessario il reindex di una singola entità, conosciuti nome della classe di modello (className) e chiave univoca (classPK)

Come spesso accade nel mondo Liferay ci sono ottimi esempi di implementazione all'interno del codice stesso del portale. In questo caso consiglio di studiare la classe com.liferay.portlet.usersadmin.util.UserIndexer che è quella utilizzata dal portale per indicizzare gli utenti e contiene ottimi spunti e punti di riferimento implementativi per tutti i metodi descritti sopra

Definizione classe indexer su liferay-portlet.xml

Oltre ad implementare l'indexer dobbiamo comunicare a liferay che la nostra portlet utilizzare un indexer. Per farlo dobbiamo inserire nel liferay-portlet.xml del nostro progetto il tag <indexer-class> con l'indicazione del nome completo (package + nome classe) del nostro indexer

```
<portlet>
    <portlet-name>test-lucene1</portlet-name>
    <icon>/icon.png</icon>
    <indexer-class>it.marcorosetti.test.search.MyEntityIndexer</indexer-class>
    <header-portlet-css>/css/main.css</header-portlet-css>
    <footer-portlet-javascript>
```

Annotation su metodi del Service

Abbiamo quasi finito, ci manca solo di invocare il processo di reindicizzazione ogni volta che una nostra entità viene creata, modificata o eliminata. Tra i vari modi che ci sono per invocare la reindicizzazione quello che consiglio è l'utilizzo dell'annotation @Indexable. Questa annotation può essere messa sui metodi dei services (tipicamente nella classe [MyEntity]LocalServiceImpl) che hanno come parametro di ritorno l'entità che vogliamo reindicizzare: in questo modo, ogni volta che un metodo annotato ritorna un'entità, Liferay si preoccupa di intercettare l'entità e di eseguirne il reindex sugli indici di Lucene.
L'annotation prevede il parametro "type", obbligatorio, che può avere 2 valori e ne configura il comportamento:

- IndexableType.REINDEX – L'entità ritornata viene reindicizzata sugli indici di Lucene

```
@Indexable(type = IndexableType.REINDEX)
@Override
public MyEntity addMyEntity(MyEntity myEntity) throws SystemException {
    myEntity.setNew(true);

    return myEntityPersistence.update(myEntity);
}
```

- IndexableType.DELETE – L'entità ritornata viene rimossa

dagli indici di Lucene

```
@Indexable(type = IndexableType.DELETE)
@Override
public MyEntity deleteMyEntity(long myEntityId)
    throws PortalException, SystemException {
    return myEntityPersistence.remove(myEntityId);
}
```

Ricerca di un'entità

La ricerca di entità avviene utilizzando principalmente le query Lucene. Si tratta di un argomento molto vasto e complesso che cercherò di affrontare in un articolo separato.

La metodologia "base" per poter eseguire una ricerca si fonda su questi oggetti e metodi

- SearchContext – contenitore di tutti i parametri di ricerca
- BooleanClause – rappresentazione di una clausola di ricerca specifica su un campo per un particolare valore
- IndexerRegistryUtil.nullSafeGetIndexer(String className) – utility per recuperare l'oggetto Indexer associato al modello che si vuole cercare
- indexer.search(SearchContext searchContext) – Metodo che esegue la ricerca vera e propria. Ritorna un oggetto di tipo com.liferay.portal.kernel.search.Hits.
- com.liferay.portal.kernel.search.Hits – Oggetto che contiene tutti i dati relativi alla ricerca e al suo risultato. Da segnalare in particolare i metodi
    - getDocs() – La lista dei risultati, eventualmente paginati, della query di ricerca. Si tratta di oggetti di tipo Document, l'oggetto specifico che utilizza Lucene come rappresentazione della nostra entità (si veda il metodo doGetDocument() dell'indexer per dettagli)
    - getLength() – Il totale dei risultati della ricerca. Da notare che, in caso di paginazione, il metodo getDocs() tornerà una sola pagina di

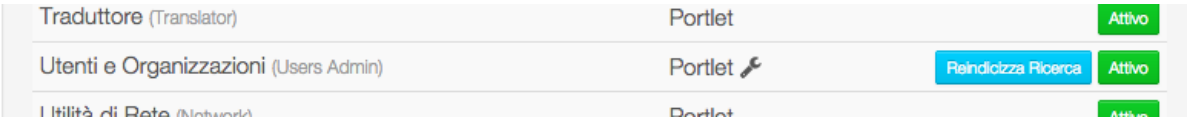
elementi, mentre getLength() tornerà il numero totale degli elementi della ricerca

```
public Hits search() throws SearchException
{
    SearchContext searchContext = new SearchContext();

    BooleanClause[] clauses = new BooleanClause[3];

    clauses[1] = BooleanClauseFactoryUtil.create(searchContext, "field1", "value 1", BooleanClauseOccur.MUST.toString());
    clauses[2] = BooleanClauseFactoryUtil.create(searchContext, "field2", "value 2", BooleanClauseOccur.SHOULD.toString());
    clauses[3] = BooleanClauseFactoryUtil.create(searchContext, "field3", "value 3", BooleanClauseOccur.MUST_NOT.toString());

    searchContext.setBooleanClauses(clauses);

    Indexer indexer = IndexerRegistryUtil.nullSafeGetIndexer(MyEntity.class);
    Hits hits = indexer.search(searchContext);
    return hits;
}
```

Reindex completo entità della portlet

Può capitare di dover reindicizzare in una volta sola tutte le entità di una portlet (ad esempio dopo una modifica batch su database). Per farlo Liferay mette a disposizione una funzionalità che lancia il reindex di una portlet su tutte le comunità configurate nel portale.

Per farlo, nel Pannello di Controllo, scheda App, cercare la portlet che si vuole reindicizzare e cliccare su Reindicizza Ricerca



Riferimenti e approfondimenti

- http://lucene.apache.org/
- https://code.google.com/p/luke/ – Tool per l'apertura e la navigazione degli indici di Lucene
- http://blog.d-vel.com/web/blog/home/-/blogs/inidicizzazione-e-ricerca-in-liferay
- http://www.scalsys.com/blog/implementing-lucene-search-in-liferay-custom-portlet/

Lucene: sincronizzarsi con l'indicizzazione

Come possiamo essere sicuri che l'indicizzazione di Lucene sia terminata?

Lucene è un motore di indicizzazione utilizzato da Liferay per la ricerca delle proprie entità e può essere facilmente implementato anche per le entità custom delle proprie portlet. Oltre agli indiscutibili vantaggi in termini di possibilità di ricerca e di velocità possiamo però incorrere in alcuni piccoli problemi o comportamenti inaspettati.

Uno di questi deriva dal fatto che il processo di indicizzazione vero e proprio viene eseguito in modo asincrono (cioè su un thread separato) rispetto al thread principale da cui parte. Questa scelta, unita ad un processo di creazione/aggiornamento dell'entità sugli indici che potrebbe richiedere molto tempo, può condurre a risultati inattesi nelle ricerche eseguite nei momenti successivi.

La soluzione a questo problema richiede non più di due righe di codice ma è utile avere una visione un po' più ampia dell'intero meccanismo di reindicizzazione per capirne meglio il funzionamento.

Un esempio per chiarire il problema

Abbiamo un'entità che, tra le altre, ha una variabile che ne identifica il colore che può assumere differenti valori(giallo, rosso, verde, etc.). Abbiamo una visualizzazione a lista che ci permette di filtrare le diverse

entità per singolo colore e per fare questo esegue una query sugli indici Lucene.

Supponiamo di avere il filtro impostato su “Giallo” e di modificare una delle entità una cambiandone il colore. Al salvataggio viene avviato da Liferay il processo di indicizzazione Lucene dell’entità ma non si attende che questo abbia effettivamente terminato l’indicizzazione prima di ritornare il controllo ai livelli superiori. In altre parole l’utente che ha fatto il salvataggio può ricevere la notifica di operazione completata prima che l’aggiornamento degli indici sia terminato. Se in questa situazione viene ricaricata la lista con il filtro a “Giallo”, l’entità incriminata continuerà a comparire anche se il salvataggio è andato a buon fine.

Il Message Bus per ottenere processi asincroni

L’asincronicità dell’indicizzazione è ottenuta da Liferay sfruttando i servizi del Message Bus. Questi servizi permettono, tra le altre cose, di inviare messaggi a destinazioni (chiamate code) senza preoccuparsi né di chi né di quando verranno ricevuti e processati.

Nel nostro caso particolare, in seguito al salvataggio, Liferay invia un messaggio sul Message Bus contenente tutti i dati utili alla reindicizzazione su una coda specifica senza attendere una risposta. Da parte di Lucene ci sarà un processo incaricato di verificare la presenza di messaggi di reindicizzazione sulla coda ed eventualmente processarli, ma su un thread separato da quello che li ha generati

Forzare l’attesa di risposta ad un

messaggio

Esiste però un meccanismo per forzare Liferay ad attendere una risposta ogni volta che invia un messaggio sul Message Bus e questo si ottiene inserendo questo codice in un punto precedente all'operazione di indicizzazione

ProxyModeThreadLocal.setForceSync(true);
Può essere una buona idea (ma va valutata a seconda dei casi specifici) inserire il codice prima di ogni Action della nostra portlet. Per fare questo è sufficiente fare l'override del metodo callActionMethod nella nostra classe che estende MVCPortlet in questo modo

```
/**
 * Portlet implementation class TestPortlet1
 */
public class TestPortlet1 extends MVCPortlet {


    @Override
    protected boolean callActionMethod(
        ActionRequest actionRequest, ActionResponse actionResponse)
        throws PortletException {

        boolean forceSync = ProxyModeThreadLocal.isForceSync();

        //marcorosetti - Ensure all thread is synchronous (lucene reindex too)  to avoid problem
        //when reloading list (before reindex process has finished)
        ProxyModeThreadLocal.setForceSync(true);
        boolean result = super.callActionMethod(actionRequest, actionResponse);

        ProxyModeThreadLocal.setForceSync(forceSync);
        return result;
    }

```

Password di accesso al database criptata

Come rendere più sicura la password di connessione al database?

Le credenziali di accesso al database vengono salvate, di default, in chiaro all'interno di un file di properties (portal-ext.properties o portal-setup-wizard.properties a seconda dei casi). Se vogliamo rendere più sicuro questo meccanismo possiamo fare una semplice modifica al codice sorgente di Liferay per poter gestire anche valori criptati.

La classe che dobbiamo modificare è com.liferay.portal.dao.jdb.DataSourceFactoryImpl, in particolare il metodo initDataSource(Properties properties):

*DataSourceFactoryImpl.java

```
82
83              tomcatDataSource.close();
84          }
85      }
86
87      @Override
88      public DataSource initDataSource(Properties properties) throws Exception {
89          Properties defaultProperties = PropsUtil.getProperties(
90                  "jdbc.default.", true);
91
92          /**
93           * marcorosetti.it - codice aggiuntivo per gestire password criptate - INIZIO
94           */
95          Enumeration<String> propEnum = (Enumeration<String>)defaultProperties.propertyNames();
96
97          while(propEnum.hasMoreElements())
98          {
99              String key = propEnum.nextElement();
100
101             if(key.equalsIgnoreCase("password"))
102             {
103                 boolean isEncrypted = GetterUtil.getBoolean(defaultProperties.getProperty("encrypted.password"));
104
105                 if(isEncrypted)
106                 {
107                     String encryptedValue = defaultProperties.getProperty(key);
108
109                     //Qui implementare la specifica funzione di decodifica
110                     String decryptedValue = new String(Base64.decode(encryptedValue));
111                     properties.setProperty(key,decryptedValue);
112                 }
113             }
114         }
115
116         /**
117          *marcorosetti.it  - codice aggiuntivo per gestire password criptate - FINE
118          */
119
120         PropertiesUtil.merge(defaultProperties, properties);
121
122         properties = defaultProperties;
```

Vediamo le operazioni importanti del codice che abbiamo inserito

1. Scorriamo tutte le proprietà impostate fino a trovare la proprietà password
2. Se è stato impostato a TRUE il valore di jdbc.default.encrypted.password allora consideriamo il valore della password come criptato e lo decriptiamo prima di sovrascriverlo nelle properties

Una volta modificato il codice dobbiamo rigenerare le librerie di liferay utilzzando i task ant messi a disposizione insieme ai sorgenti: possiamo utilizzare compile e poi andare a sostituire la classe generata all'interno del file **portal-impl.jar** nella cartella **tomcat-7.0.42\webapps\ROOT\WEB-INF\lib** oppure rigenerare l'intero file con il task jar.

Alcune considerazioni

- Nell'esempio riportato l'algoritmo di cript-decrypt è un semplice Base64 hashing che serve solo come "placeholder". Possiamo ovviamente complicare come vogliamo l'algoritmo a seconda delle nostre esigenze
- L'approccio applicato alla proprietà "password" si può applicare a qualsiasi proprietà si intenda criptare
- Se non facciamo altre modifiche ai sorgenti potrebbero venire stampati nei log dei messaggi a livello WARN del tipo "*Property encrypted.password is not a valid…*". Possiamo tranquillamente ignorare questi messaggi
- L'esempio vale per Liferay versione 6.2
- La sicurezza di questo approccio potrebbe essere aumentata andando ad offuscare il codice della classe
- Un approccio di sicurezza più architetturale consiste nell'utilizzare datasource esterni e iniettati tramite nome JNDI. In questo modo la gestione della sicurezza delle credenziali di accesso è demandata all'application server