UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di laurea triennale in Ingegneria Informatica
TESI DI LAUREA

IMPLEMENTAZIONE DI UN CONTENT REPOSITORY CON LE SPECIFICHE JCR E APACHE JACKRABBIT: UNA NUOVA SOLUZIONE PER LA GESTIONE DOCUMENTALE

Laureando: Marta Tombolato

Relatore: Dott. Giorgio Maria Di Nunzio

26 Aprile 2010

Anno Accademico 2009-2010

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei listati codice

Sommario

Negli ultimi anni grazie alla crescita del numero delle applicazioni e alla diffusione dei contenuti generati dagli utenti, si è riscontrato il problema della gestione di un insieme di informazioni numeroso ed eterogeneo. Basti pensare ad un server Web che gestisce una mole consistente di dati dove si presentano documenti multimediali (ad esempio recensioni, video, immagini, PDF) e altre informazioni tutte di natura diversa. Una semplice base di dati relazionale non è sufficiente a fornire il supporto necessario per questo tipo di applicazioni.

L'obiettivo della tesi sarà quello di definire un possibile modello di sviluppo per la gestione di informazioni di natura complessa attraverso l'uso di un archivio di informazioni chiamato Content Repository. Esso nasce dall'interesse di innovazione e ricerca dell'azienda Sanmarco Informatica S.p.a., sede del tirocinio, la quale desiderava studiare le funzionalità di un Content Repository ed in seguito concretizzare lo studio, implementando il Repository per la gestione documentale aziendale. La motivazione principale che ha indotto Sanmarco Informatica a studiare una struttura diversa dal database relazionale è il fatto che un Content Repository permette di memorizzare tipologie di dati eterogenee.
Mediante l'uso di un Content Repository si ha una struttura dinamica e gerarchica formata da nodi con le rispettive proprietà, che sostituiscono le tabelle e gli attributi, dove questi ultimi potranno essere integrati senza cambiare la struttura del nodo. Con una struttura dinamica si avranno notevoli vantaggi come l'aggiunta di una nuova proprietà a distanza di tempo soltanto nell'unico nodo necessario, non ci saranno più relazioni fra attributi ma relazioni fra nodi in modo gerarchico, non ci potranno essere riferimenti pendenti, sarà rispettata così l'integrità referenziale. Non si è più legati quindi ad una struttura relazionale e statica, dove si ha l'obbligo di usare tabelle fisse.

Un'altra motivazione è data dalla possibilità di sfruttare la ricerca 'full text' su ciascun documento e sulle sue rispettive proprietà. Ogni singola parola contenuta nel documento, infatti, nel momento in cui è presente nel Content Repository è automaticamente indicizzata. Questa risulta essere la motivazione più importante per Sanmarco Informatica. Poter ricercare nel contenuto dei documenti, anche in lingua non italiana, e poter ottenere come risultato il documento corretto, ma soprattutto il contenuto del documento senza imperfezioni con il riconoscimento di tutti i caratteri. Il Content Repository permette l'indicizzazione dei caratteri di qualsiasi lingua.

Per poter rendere concrete le motivazioni sopra riportate si è scelto Apache Jackrabbit, prodotto che implementa la logica della specifica JSR-170 nota come

JCR (Java Content Repository). Apache Jackrabbit è un progetto Open Source, multipiattaforma e stabile, inoltre, è l'implementazione di riferimento della specifica JCR; queste sono le ragioni che hanno portato l'azienda ad implementare il proprio Content Repository con Jackrabbit.

Esistono altri prodotti molto noti sul mercato che si basano su una soluzione di Content Repository in pacchetti già strutturati. Apache Jackrabbit, a differenza di questi prodotti già sviluppati e pronti all'uso, è il Content Repository 'per definizione', su cui poter iniziare ad implementare il Repository su misura per l'azienda.

Capitolo 1

Introduzione

L'aumento delle applicazioni e della diffusione dei contenuti generati dagli utenti, ha portato a gestire un insieme di informazioni numeroso e di natura differente. Si può pensare ad un portale Web che gestisce una quantità elevata di informazioni, quali documenti multimediali e altri dati eterogenei.
Un semplice database relazionale non fornisce in maniera sufficiente il supporto necessario per gestire applicazioni e contenuti generati dagli utenti contenenti un insieme di informazioni numeroso e allo stesso momento di natura differente. In un database relazionale si ha una struttura statica formata da tabelle, a sua volta composte da righe contenenti i record e colonne contenenti ciascun campo dei record. Ovviamente la struttura tabellare deve essere creata prima di immagazzinare i dati, questo risulta un limite in quanto se nel tempo si ha il bisogno di aggiungere un ulteriore attributo presente in un unico record lo si deve inserire in tutta la tabella. Ci sarebbe la possibilità di creare dei record dinamici, i quali contengono delle proprietà aggiuntive rispetto agli altri, ma il tempo per analizzarli risulta decisamente elevato.
Con un Content Repository questo limite viene eliminato.
Prima di tutto non ci sono più le tabelle ma si ha una struttura gerarchica. Una struttura dinamica dove i dati vengono organizzati in una struttura ad albero. La flessibilità di questa soluzione consiste nel vedere le informazioni come nodi e proprietà. Un nodo è un oggetto, di natura indefinita, ed una proprietà è un informazione concreta. Quindi nel momento in cui si ha la necessità di inserire dei dati viene creato un nodo (paragonabile ad un record) con delle proprietà (attributi).

Come si può vedere in Figura 1.1, la radice ha tre oggetti, a, b e c. A loro volta questi oggetti contengono altri oggetti fino ad arrivare alle informazioni (foglie). Attraverso questa semplice astrazione si può pensare di immagazzinare informazioni strutturate, senza una struttura fissata a priori (tipico problema dei database relazionali) [1].

Infatti se a distanza di tempo si desidera inserire una nuova proprietà è possibile aggiornare il singolo nodo e non l'intera struttura.

Inoltre nel Content Repository sono garantite l'integrità di entità, l'integrità referenziale, e l'integrità di dominio. L'integrità di entità è garantita dal fatto

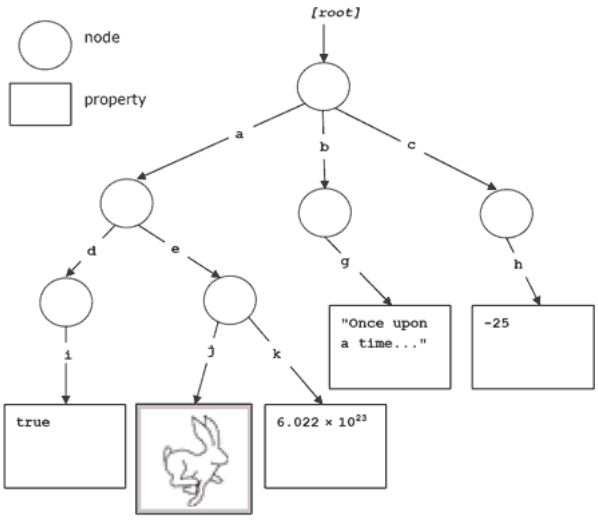

Figura 1.1: Struttura JCR.

che, ogni nodo è unico e identificato dal suo UUID. L'integrità referenziale è garantita dalla proprietà referenceable presente nel nodo. Un nodo referenceable non può essere eliminato finché tale proprietà è presente, per eliminare tale nodo è necessario togliere il vincolo [2]. Questa è una maggiore sicurezza in quanto non sarà mai possibile avere riferimenti pendenti. L'integrità di dominio può essere garantita assegnando ai nodi proprietà specifiche che contengono i valori in intervalli predefiniti.

Per definire in maniera semplice un Content Repository, lo si può pensare come un grosso contenitore di informazioni: nessuna struttura predefinita, semplicemente un sistema capace di immagazzinare e recuperare in maniera ottimizzata informazioni, anche relazionate tra loro. La semplicità di questa definizione non inganni, in quanto si vedrà in seguito che, a fronte di 'semplici' funzionalità di lettura e scrittura, la gestione del sistema Content Repository nasconde all'utente finale un'architettura astratta per poter dare la flessibilità di archiviare diversi tipi di dati. In ambiente Java la scelta di trovare una soluzione standard al problema iniziò con le specifiche JSR-170 [3] note anche come JCR. Il Content Repository API per Java (JCR) è una specifica per accedere in modo uniforme ai contenuti archiviati. I Content Repository sono usati nei sistemi di gestione dei contenuti, CSM (Content Management System), per conservare dati e metadati. In seguito alla nascita di queste specifiche, si diede l'avvio alla creazione di uno strato software astratto in modo da consentire allo sviluppatore di applicazioni di non preoccuparsi di come il prodotto Content Repository fosse implementato a più basso livello. Grazie alla possibilità di immagazzinare qualsiasi tipo di dato, il Content Repository risulta un utile strumento alle limitazioni dei database relazionali utilizzati per la gestione dei documenti [1].

Il Content Repository, quindi, è un contenitore di informazioni per l'archiviazione, la ricerca e per il recupero di dati gerarchici, utilizzato per gestire e conservare dati e metadati. Il JCR nasce dalla necessità di gestire sistemi di contenuti che richiedono la memorizzazione dei documenti e altri oggetti binari

con i metadati associati. Oltre all'archiviazione dei dati, il Content Repository prevede il controllo delle versioni, le transazioni, le modifiche ai dati, l'importazione o l'esportazione di dati in formato xml. Nel JCR, le informazioni vengono memorizzate come un albero di nodi con le rispettive proprietà. I dati vengono, inoltre, immagazzinati nelle proprietà, le quali possono contenere valori semplici come stringhe e numeri o dati binari di lunghezza arbitraria. I nodi possono puntare ad altri nodi attraverso una proprietà speciale di riferimento. In questo modo i nodi presenti nel JCR garantiscono l'integrità e l'ereditarietà. I tipi di nodo aggiuntivi includono il tipo di nodo 'referenceable' che permette all'utente di relazionare nodi mediante l'identificatore universale univoco (UUID). Un altro tipo di nodo conosciuto è il 'versionable', questo tiene traccia nel repository della storia del documento immagazzinando una copia di ogni versione. Un Content Repository può essere interrogato con query XPath, è preferibile usare questo linguaggio in quanto viene interrogata una struttura gerarchica. Sono ammesse anche query SQL. Apache Jackrabbit è l'implementazione di riferimento del Content Repository, supporta anche l'integrazione del motore di ricerca Apache Lucene per effettuare ricerche full text nei contenuti delle proprietà e dei documenti.

1.1 Problemi aziendali che portano all'uso di un nuovo archivio dati

Sanmarco Informatica azienda con sede a Grisignano di Zocco (Vicenza) nata negli anni '80 come software house specializzata negli applicativi per aziende manufatturiere, si è evoluta in un crescendo di esperienze di successo e di scelte imprenditoriali che individuano nella specializzazione del capitale umano l'elemento centrale [4].

L'interesse di innovazione e ricerca di Sanmarco Informatica ha portato ad iniziare lo studio di un nuovo archivio dati.
L'azienda, a causa di alcuni problemi legati all'uso dei database relazionali usati nella gestione documentale, desidera iniziare a studiare un modello di memorizzazione e archiviazione dati diverso e punta l'attenzione sul Content Repository, il quale, come già precedentemente analizzato, offre molteplici funzionalità.
Le motivazioni della scelta sono principalmente tre:

- Possibilità di memorizzare dati eterogenei
- Possibilità di effettuare ricerche full text
- Possibilità di avere una struttura 'dinamica'

Un Content Repository permette di memorizzare dati di natura diversa: eterogenei. Con ciò si intende che una proprietà di un nodo può contenere un dato di natura testuale, numerica, data-ora ma anche binario. Infatti tutto il contenuto di un documento viene immagazzinato in formato binario.

La possibilità di archiviare un documento in questo modo permette di sfruttare le ricerche full text sul documento. Quest'ultimo infatti viene indicizzato.

Sanmarco Informatica necessitava di ricercare nel contenuto di un file, anche in testi non in lingua italiana, e ricavare il testo codificato in UTF-8 correttamente. Il Content Repository permette di effettuare ricerche full text ed indicizza qualsiasi tipo di carattere, operazione non fattibile nelle applicazioni della gestione documentale presente in azienda.

Per gestione documentale, Document Management System (DMS) [5], si intende una categoria di applicativi che serve ad organizzare e facilitare la creazione di documenti e di altri contenuti. Tecnicamente il DMS è un'applicazione che si occupa di eseguire operazioni sui documenti, catalogandoli ed indicizzandoli secondo determinati algoritmi. Gli applicativi che compongono questi prodotti permettono di catalogare, organizzare, spedire, scansionare e archiviare qualsiasi tipo di documento elettronico e non. La riduzione degli spazi necessari, l'abbattimento di costi di gestione, l'inalterabilità dei documenti, la riservatezza, l'efficienza del lavoro e il rispetto dell'ambiente sono solo alcuni vantaggi forniti dalla gestione documenti.

Infine, Sanmarco Informatica desiderava poter inserire una singola proprietà in un nodo. Grazie al Content Repository non si è più legati ad una struttura relazionale e statica, dove si ha l'obbligo di usare tabelle fisse che contengono record con attributi che non mutano, ma come illustrato nella Figura 1.1 si ha una struttura ad albero dinamica dove si ha la possibilità di inserire una singola proprietà in un nodo.

1.2 Implementazioni disponibili

Attualmente, nel mercato, sono già presenti delle applicazioni che implementano Content Repository.
In seguito ne verranno presentati brevemente alcuni.

1.2.1 Alfresco

Alfresco [6] è un CSM (Content Management System) per sistemi Windows e Linux/Unix.

Letteralmente un CSM [7] è un 'sistema di gestione dei contenuti', uno strumento software installato e studiato per facilitare la gestione dei documenti, svincolando l'amministratore da conoscenze tecniche di programmazione.

Alfresco offre due versioni: la prima Community Edition, software libero rilasciato sotto la licenza Open Source GNU/PGL. La seconda Enterprise Edition, versione commerciale con maggiori funzionalità.

Il suo design è orientato verso gli utenti che richiedono un elevato grado di modularità e prestazioni scalabili. Alfresco comprende un repository di contenuti, un 'out-of-the-box' portale web per la gestione e l'utilizzo standard del contenuto del portale, un'interfaccia di file che fornisce la compatibilità del sistema in Microsoft Windows e Unix, un sistema di gestione dei contenuti web in grado

di virtualizzare siti statici tramite Apache Tomcat e il motore di ricerca Lucene. Alfresco è stato sviluppato utilizzando Java, JSP e JavaScript.

1.2.2 Apache Jackrabbit

Apache Jackrabbit [8] è un Content Repository Open Source per la piattaforma Java. Il progetto Jackrabbit iniziò nell'agosto del 2004 quando Day Software concesse la licenza d'uso di una prima applicazione delle API Java Content Repository (JCR). Jackrabbit viene anche utilizzato come implementazione di riferimento delle specifiche JSR-170. Ora è un progetto di alto livello della Apache Software Foundation. Il JCR è un'API specifica per sviluppatori di applicazioni che utilizzano l'interazione fra repository di contenuti, i quali forniscono servizi come la ricerca, il versionamento e molto altro ancora.

Le caratteristiche principali sono: contenuti gerarchici struttutati o non strutturati, tipi di nodi e di proprietà diversi, Query XPath o SQL per interrogare il Content Repository, controllo degli accessi, revisioni di documenti, integrità referenziale.

1.2.3 Hippo CSM

Hippo CSM [9] è un Content Management System. Tale progetto è stato avviato e gestito da Hippo, è mirato a gestire organizzazioni medio-grandi di contenuti per la distribuzione multi canale come siti web o intranet. Esso è un modo facile e flessibile per usare i dati.

Hippo è di facile uso e si basa sul progetto Open Source Apache Jackrabbit.

1.2.4 Magnolia

Come Hippo CSM, Magnolia [10] è un CSM Open Source basato su Apache Jackrabbit. Magnolia è favorito per la sua semplicità d'uso e la disponibilità sotto una licenza Open Source.

1.3 Obiettivi e requisiti del progetto

Lo sviluppo di Content Repository finalizzati all'archiviazione dei dati è oggi un argomento in continua evoluzione, ma soprattutto in continua standardizzazione. Questo progetto si propone di studiare, progettare e realizzare un Content Repository per la gestione documentale nell'ambito dell'azienda Sanmarco Informatica.

L'obiettivo principale del tirocinio era lo studio approfondito e dettagliato della specifica JCR (Java Content Repository).

Era necessario capire come il JCR gestisce i dati e i metadati; essendo esso un database object-oriented era utile comprendere in che modo tratta le relazioni

dei database relazionali e, una volta studiate le specifiche, implementare un repository per la memorizzazione dei documenti, collegandolo in seguito al sistema di gestione Galileo, in questo caso alla parte documentale.

La voglia di investire in tale progetto, per l'azienda, significava aggiungere potenzialità al proprio software documentale come la ricerca full text, l'aggiunta di nuovi attributi nel tempo, quindi la possibilità di personalizzare un progetto già costruito e la possibilità di memorizzare dati di natura diversa, visto che il contesto attuale ne propone di svariati tipi.

La progettazione del Content Repository ha tenuto conto dei seguenti requisiti:

- **Soluzione Open Source:** l'implementazione del repository si è basata sulla scelta di applicazioni di autori (meglio i detentori dei diritti) che ne permettono, anzi ne favoriscono il libero studio e l'apporto di modifiche da parte di altri programmatori.
- **Soluzione stabile:** far sì che il repository sia implementato con applicativi che non vengono modificati giornalmente o quasi, ovvero che non è previsto che vengano rivoluzionati del tutto da un giorno all'altro.
- **Soluzione multipiattaforma:** si è cercato di scegliere una soluzione che fosse quanto più possibile multipiattaforma e aderente agli standard, utilizzabile da qualunque terminale senza il bisogno di 'apposite' configurazioni per il sistema se non quel minimo di requisiti comuni alla maggior parte delle piattaforme. Infatti il repository creato potrà essere posto in un sistema Windows, Linux/Unix o Macinthos.
- **Soluzione generalizzata:** nelle classi Java create per lo sviluppo del Content Repository si è scelto di parametrizzare tutti i possibili parametri in ingresso, in modo da vedere il Content Repository a lunga distanza, quando non viaggerà più in parallelo con Galileo ma vivrà di vita propria.

L'azienda sceglie Apache Jackrabbit, prima di tutto perché è il Java Content Repository 'per definizione', non dipende da un'implementazione di riferimento. Inoltre, sia Hippo CSM che Magnolia sono costruiti su JCR API e usano Jackrabbit come repository di default.
Apache Jackrabbit non ha una struttura predefinita da rispettare come Alfresco, Hippo CMS e Magnolia, i quali sono dei prodotti già strutturati pronti da installare.
Sanmarco Informatica desiderava implementare un proprio archivio dati, quindi anch'essa utilizzando Apache Jackrabbit come repository di default implementerà un Content Repository specifico per la gestione documentale sfruttando tutte le caratteristiche principali di Jackrabbit elencate in precedenza.

Il Capitolo 2 propone una breve panoramica sulle motivazioni generali della nascita dei Content Repository e una rapida presentazione della specifica Java

Content Repository. Nel Capitolo 3 verrà presentato uno studio dettagliato delle specifiche JSR-170 andando ad analizzare ciò che viene offerto dai tre livelli, soffermandosi anche su alcuni metodi presenti nelle specifiche in esame. Il Capitolo 4 spiegherà la scelta implementativa, ponendo anche l'attenzione sulle funzionalità fornite e sulla configurazione. Nel Capitolo 5 sarà illustrato il progetto concreto, svolto in azienda, utilizzato per la migrazione da Galileo al JCR, riguardante la gestione documentale. E' da tenere presente che per il momento le due applicazioni lavorano in parallelo. Verranno fatte delle considerazioni finali e citati i possibili sviluppi futuri nel Capitolo 6.

Capitolo 2

Il modello Repository

2.1 Dai database relazionali ai Content Repository

Il termine database [11] indica un 'archivio strutturato' in modo tale da consentire l'accesso e la gestione dei dati stessi (l'inserimento, la ricerca, la cancellazione ed il loro aggiornamento) da parte di particolari applicazioni software ad essi dedicate. Il database è un insieme di informazioni, di dati che vengono suddivisi per argomenti in ordine logico a loro volta suddivisi per categorie.
Informalmente e impropriamente, la parola database viene spesso usata come abbreviazione dell'espressione Database Management System (DBMS), che invece si riferisce a una vasta categoria di sistemi software che consentono la creazione e la manipolazione (gestione) efficiente dei dati di un database.

La base di dati, oltre ai dati veri e propri, deve contenere anche le informazioni sulle loro rappresentazioni e sulle relazioni che li legano. Spesso, ma non necessariamente, una base dati contiene: strutture dati che velocizzano le operazioni frequenti, tipicamente a spese di operazioni meno frequenti; collegamenti con dati esterni, cioè riferimenti a file locali o remoti non facenti parte del database; informazioni di sicurezza, che autorizzano solo alcuni profili utente ad eseguire alcune operazioni su alcuni tipi di dati; programmi che vengono eseguiti, automaticamente o su richiesta di utenti autorizzati, per eseguire elaborazioni sui dati.

In un sistema informatico, una base di dati può essere manipolata direttamente dai programmi applicativi, interfacciandosi con il sistema operativo (file system). Tale strategia era quella adottata universalmente fino agli anni sessanta, ed è tuttora impiegata quando i dati hanno una struttura molto semplice, o quando sono elaborati da un solo programma applicativo.

A partire dalla fine degli anni Sessanta per gestire basi di dati complesse condivise da più applicazioni si sono utilizzati appositi sistemi software, detti sistemi per la gestione di basi di dati (in inglese Database Management System o DBMS). Uno dei vantaggi di questi sistemi è la possibilità di non agire direttamente sui dati, ma di vederne una rappresentazione concettuale.

La ricerca nel campo delle basi di dati studia la progettazione di queste e l'implementazione di DBMS. Interpreta e analizza i dati contenuti nei database.

Le basi di dati possono avere varie strutture, tipicamente, in ordine cronologico:

- **gerarchica:** rappresentabile tramite un albero - anni sessanta.
- **reticolare:** rappresentabile tramite un grafo - anni sessanta.
- **relazionale:** (attualmente il più diffuso) rappresentabile mediante tabelle e relazioni tra esse - anni settanta.
- **ad oggetti:** estensione alle basi di dati del paradigma Object Oriented, tipico della programmazione a oggetti - anni ottanta.
- **semantica:** rappresentabile con un grafo/albero relazionale - inizio anni duemila.

Le necessità attuali [12] richieste ad un archivio dati sono:

- **Immagazzinare dati eterogenei:** Archiviazione di dati di natura diversa.
- **Ricerche full text:** Indicizzazione dell'intero contenuto di un documento per effettuare ricerche in esso.
- **Revisioni di un documento:** Implementazione di meccanismi di versioning e possibilità di tener traccia delle modifiche.

Per poter rispondere alle tre necessità esposte in precedenza vennero creati i Content Repository. Sono stati immessi nel mercato alcuni CSM, ma ognuno presentava una propria versione di Content Repository, e ogni fornitore doveva fornire le proprie API per interagire con il repository. Questo era il problema di fondo. Studiare una particolare API e potenzialmente legare il codice ad un'esclusiva applicazione CSM, rappresentava una difficoltà per gli sviluppatori di applicazioni ed inoltre uno spreco di risorse per le aziende.

Nel 2002 [13], venne definita un'API standard: la specifica JSR-170 (JCR - Java Content Repository) o Content Repository for Java Technology. Essa viene definita come 'uno standard, un modo di implementare in maniera indipendente l'accesso a contenuti all'interno di un Content Repository' e va a definire un Content Repository come 'un alto livello di gestione delle informazioni'. Java Content Repository API (JSR-170) è un tentativo di rendere standard delle API che possono essere utilizzate per accedere ad un repository, quindi venne risolto il problema di studiare particolari API per ciascun CSM. Con JSR-170 si sviluppa il codice utilizzando solamente le classi e le interfacce presenti nel package javax.jcr (strato software astratto che consente allo sviluppatore di non preoccuparsi di come il Content Repository sia implementato a più basso livello). Tale pacchetto è in grado di lavorare con qualsiasi Content Repository. L'implementazione di riferimento delle specifiche JSR-170 è Apache Jackrabbit. Poichè il numero di fornitori di Content Repository è aumentato la necessità di utilizzare un'interfaccia di programmazione comune risultava evidente, è proprio questa motivazione

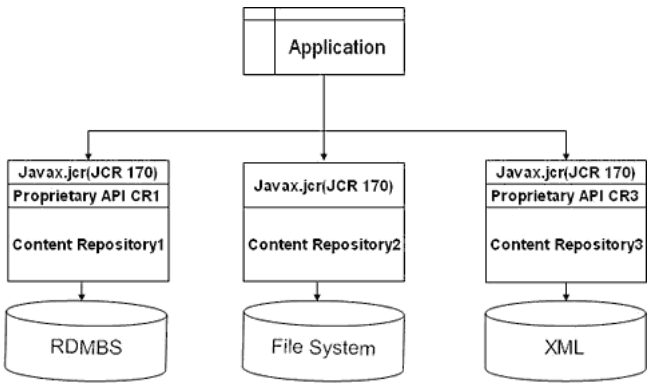

Figura 2.1: Gestione astratta e unificata di fonti di dati differenti.

che fa entrare in gioco le JSR-170. Esse offrono un'interfaccia di programmazione utilizzata per la connessione al Content Repository. Inoltre permettono di sviluppare un proprio programma indipendentemente dal Content Repository utilizzato. Ne è di esempio la Figura 2.1 (l'immagine indica i possibili modelli di persistenza, ma il problema è esteso anche all'eterogeneità delle informazioni in gioco).

Lo scopo principale della nascita delle JSR-170 [1] è stato quello di trovare un semplice modello su cui basare le implementazioni dei diversi Content Repository presenti nel mercato. Il problema di fondo era quello di trovare un'interfaccia comune, che fungesse da wrapper, ai diversi modelli di logica e persistenza per i prodotti nuovi o quelli già esistenti.

Le JSR-170 definiscono la struttura di un Content Repository ad albero come illustrato nel Capitolo 1. Inoltre, definiscono anche i livelli di conformità quali Livello 1: lettura del repository, Livello 2: scrittura nel repository e Blocchi funzionali contenenti operazioni aggiuntive: operazioni avanzate nel repository.

2.2 JCR API - JSR-170

Tutte le operazioni più richieste sono definite in queste specifiche, alcune funzionalità, ad esempio di amministrazione, non sono ancora ben definite. I livelli come già citato precedentemente sono tre [14].

- **Livello 1:** Lettura del repository. Figura 2.2.

  Il primo livello copre un gran numero di applicazioni semplici come la ricerca negli archivi di dati e la possibilità di leggere informazioni dal repository. Permette l'accesso in sola lettura ai contenuti memorizzati nel repository.
  Il primo livello è finalizzato a consentire agli sviluppatori e a chi fa uso di un Content Repository di effettuare operazioni di ricerca e di analisi del contenuto.

- **Livello 2:** Scrivere nel repository. Figura 2.3.

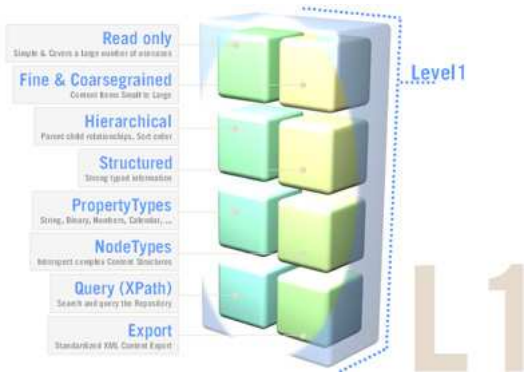

Figura 2.2: Livello 1 JSR-170.

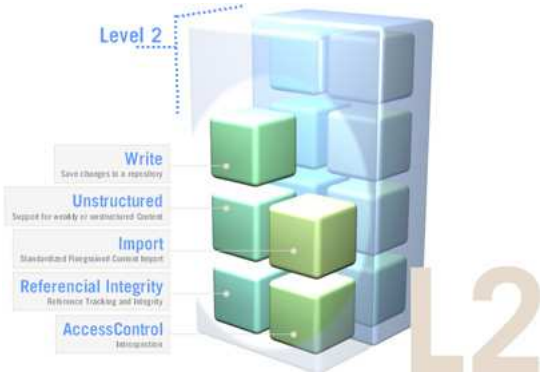

Figura 2.3: Livello 2 JSR-170.

Il secondo livello specifica le funzionalità di scrittura in modo da interagire in maniera più profonda con il repository.
Le funzionalità del Livello 2 implicano la gestione e la generazione di dati e informazioni strutturate e non, da inserire direttamente nel repository.

- **Blocchi funzionali contenenti operazioni aggiuntive:** Figura 2.4.

  Sopra il primo e secondo livello esistono dei blocchi funzionali con prestazioni avanzate da poter utilizzare nel repository, come ad esempio il versionamento.

Poiché il numero di fornitori che offrono Content Repository è aumentato, la necessità di avere un'interfaccia comune a livello di programmazione è diventato evidente. L'obiettivo della specifica API Content Repository for Java Technology è quello di fornire questo tipo di interfaccia e, in tal modo, porre le basi per una vera industria a livello di infrastrutture di contenuti. Gli sviluppatori di applicazioni e gli sviluppatori di soluzioni personalizzate volevano essere in grado di evitare i costi connessi all'apprendimento di una particolare API fornita da ciascun produttore di repository. Con la specifica JSR-170, i programmatori saranno in grado di sviluppare Content Repository basati su una logica di applicazione indipendente dall'architettura dell'archivio sottostante, detto anche archivio fisico.

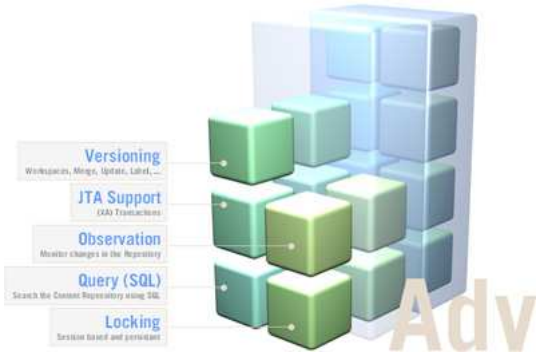

Figura 2.4: Blocchi funzionali JSR-170.

I clienti potranno, inoltre, scambiare i loro archivi di base senza toccare nessuna delle applicazioni costruite su di essi.

Gli elementi del repository Un Content Repository è costituito da una o più aree di lavoro, 'Workspace', contenenti un albero di oggetti. Un elemento può essere un nodo o una proprietà. Ogni nodo può avere zero o più nodi figli ciascuno dei quali contiene zero o più proprietà. Vi è un unico nodo radice (root node) che non ha padre. Tutti gli altri nodi hanno un genitore. Le proprietà sono presenti se esiste il nodo, infatti sono gli attributi di questo, rappresentano le foglie dell'albero. Tutto il contenuto effettivo del repository viene memorizzato all'interno dei valori delle proprietà.

2.3 API nozioni di base

Il repository nel suo complesso è rappresentato da un oggetto Repository. Un client si connette al repository chiamando

Repository.login

opzionalmente specificando un workspace e le credenziali, 'Credentials', (username e password). Il cliente riceve quindi un oggetto di tipo sessione, 'Session' nel workspace specificato. Attraverso l'oggetto Session il cliente può accedere a qualsiasi nodo o proprietà o semplicemente alla struttura del workspace. Le API forniscono metodi sia per attraversare la struttura dell'albero sia per accedere ad un particolare elemento.

2.3.1 Accedere al repository

L'accesso alla struttura del repository è un'operazione fondamentale e di base contenuta a Livello 1 delle JSR-170.

L'accesso inizia con

Node Session.getRootNode()

Questo comando restituisce il nodo radice, si può accedere ai nodi dei livelli successivi con

Node Node.getNode(String relPath)

dove relPath è il path relativo. Un modo simile per ottentere una data proprietà è

Property Node.getProperty(String relPath)

Con

String Property.getString()

si può accedere direttamente alle informazioni contenute nelle proprietà, oppure sottoforma di oggetto Value usando

Value Property.getValue()

in seguito il dato Value può essere visualizzato come stringa con un

Value.getString()

Un metodo di accesso diretto è

Node Session.getNodeByUUID(String uuid)

può essere usato per accedere ad un determinato nodo mediante l'identificativo univoco universale (Universally Unique Identifiers UUID).

2.3.2 Scrivere nel repository

Scrivere nel repository è un'operazione che sta a Livello 2 delle JSR-170.

Dopo aver acquisito una Session, il client può scrivere nel repository aggiungendo o rimuovendo nodi e proprietà o cambiando semplicemente i valori. Per aggiungere un nodo dopo aver ottenuto il nodo radice, basta editare

Node newNode = rootNode.addNode(NewNode)

per settare una proprietà al nodo creato

newNode.setProperty(PropertyName, PropertyValue)

Ovviamente è necessario salvare qualsiasi modifica con

Session.save()

altrimenti le modifiche non saranno archiviate nel repository.

2.3.3 Rimuovere nodi e proprietà

Per eliminare un elemento viene utilizzato il metodo

myNode.remove()

Per rimuovere una singola proprietà da un nodo, è possibile scegliere la rimozione sia del nome che del valore o solamente impostare a null il valore.

2.4 Immagazzinare nella sessione transitoria

Il metodo

Session.save()

è necessario perché le modifiche apportate non sono immediatamente immagazzinate nel workspace. Le modifiche sono tenute in un deposito temporaneo associato all'oggetto finché non viene invocato tale metodo che le renderà permanenti. Se si desidera scartare le modifiche si invocherà

Session.refresh(false)

Gli aggiornamenti non ancora salvati sono chiamati 'modifiche in sospeso'. Quando viene invocato il salvataggio tutte queste modifiche vengono rese visibili a tutte le sessioni, il refresh, invece elimina tutte le modifiche presenti nell'oggetto Session.

2.4.1 Elementi: nodi e proprietà

I nodi e le proprietà hanno delle funzionalità comuni, i metodi comuni sono definiti nell'interfaccia Item, la quale propone metodi alle sotto-interfacce Node e Property. Il diagramma UML della Figura 2.5 chiarisce le relazioni che esistono fra queste interfacce.

Il diagramma indica che Node e Property sono sotto-interfacce di Item. Una singola Property ha uno e soltanto un Node genitore. Un Node può avere nessun genitore, se è il nodo radice, o un genitore. Un Node può avere qualsiasi numero di oggetti Item figli, vale a dire sia proprietà che nodi.

2.4.2 Livelli di conformità

Questa specifica è divisa, come già visto, in tre parti: Livello 1, Livello 2 e Blocchi funzionali contenenti operazioni aggiuntive. Le funzioni sono divise nel seguente modo:

- Il Livello 1 comprende:
  - Recupero e attraversamento dei nodi e delle proprietà.
  - Lettura dei valori delle proprietà.

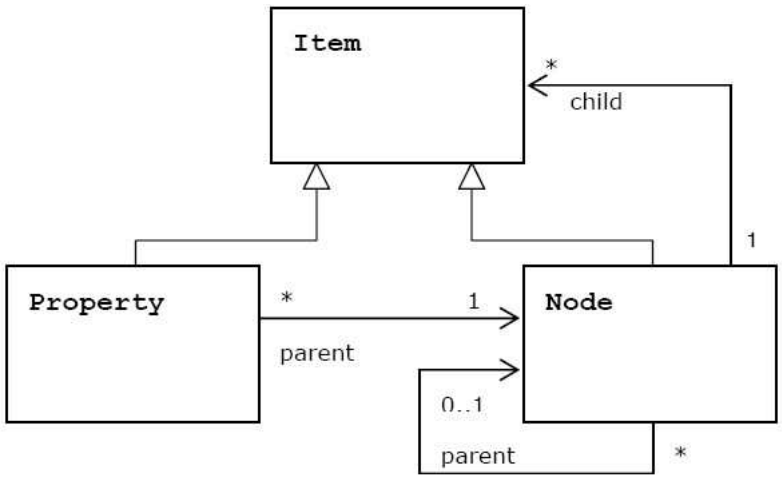

Figura 2.5: Diagramma UML - Items: Node and Property.

  - Rimappatura del namespace, se necessario.
  - Esportazione in XML.
  - Query con sintassi XPath.
  - Scoperta dei nodi disponibili.
  - Scoperta dei permessi di accesso.

- Il Livello 2 aggiunge:
  - Aggiunta e rimozione di nodi e proprietà.
  - Scrittura dei valori delle proprietà.
  - Cambiamenti dei namespace.
  - Importazione da XML.
  - Assegnazione di un tipo di nodo ai nodi.

- I Blocchi funzionali contenenti operazioni aggiuntive comprendono:
  - Transazioni.
  - Versioni.
  - Osservazione degli eventi.
  - Blocco
  - Query con sintassi SQL.

2.5 Elementi: uguaglianza di nomi e ordinabilità

2.5.1 Nodi e proprietà

Ci sono alcuni casi dove un nodo può riportare il nome di uno stesso nodo, ma questo non risulta un problema in quanto il nodo viene riconosciuto univocamente

dal suo UUID. Quindi per ricercare un nodo con lo stesso nome basta passare il path del nodo corretto al metodo

Node.getNode(relPath)

il quale itera tutti i nodi con quel relativo path. Un nodo con lo stesso nome avrà in più una notazione del tipo *[numero nomi uguali]* accanto al nome. Ad esempio il percorso

/a/b[2]

specifica che nel path

/a/b

esistono due nodi di nome b. Ovviamente se sarà presente un unico nome il path sarà

/a/b

equivalente a

/a[1]/b[1]

in questo caso gli indici vengono omessi.

I nodi possono essere ordinabili mediante il metodo

Node.orderBefore

in questo modo verranno ordinati all'interno del repository una volta salvati. Se non viene riportato questo metodo i nodi seguiranno l'ordine in cui sono stati riposti nel repository.

A differenza dei nodi le proprietà non possono avere lo stesso nome, ma possono essere multi-valore.
Le proprietà non potranno mai essere ordinate.

2.6 Namespace

Il nome di un nodo o di una proprietà contiene un prefisso delimitato da due punti, carattere che indica il namespace del nodo o della proprietà. I namespace prefissati sono:

- **jcr:** Namespace riservato alle proprietà che vengono create automaticamente come ad esempio jcr:UUID.
- **nt:** Riservato ai tipi di nodo primitivi, ad esempio nt:file.
- **mix:** Riservato per i nodi di tipo mixin, come mix:versionable.

A Livello 1 vengono usati i namespace già esistenti nel repository, a Livello 2 è possibile invece creare un nodo personalizzato e quindi creare un proprio namespace.

2.7 Proprietà

Una proprietà può essere di tipo: String, Binary, Date, Long, Double, Boolean, Name, Path, Reference. A Livello 1 sono forniti i metodi per leggere le proprietà, a Livello 2 per scriverle.

2.7.1 Proprietà multi-valore

In alcuni casi una proprietà può avere più di un valore. Una proprietà che può avere più di un valore viene indicata come multi-valore. I valori sono classificati in ordine. L'accesso ai valori di una proprietà multi-valore si effettua con

Property.getValues

il quale restituisce un array di oggetti Value contenenti i valori nell'ordine prescritto. Accedere ad una proprietà multi-valore con il metodo

Property.getValue

o ad una proprietà a valore unico con

Property.getValues

porterà ad un'eccezione del tipo

ValueFormatException

I valori memorizzati in una tale proprietà sono tutti dello stesso tipo.

2.7.2 Tipi di proprietà reference, path e name

Una proprietà viene memorizzata col tipo NAME quando è necessario archiviare tipi di stringhe che rappresentano namespace registrati. Ne può essere d'esempio

setProperty("nomeFile", nt:file)

La proprietà PATH rappresenta un path in un workspace e può anche essere usato per riferirsi ad altri workspace. Questa proprietà però non garantisce l'integrità referenziale, in altre parole può puntare ad una locazione che non esiste.

Una proprietà REFERENCE viene utilizzata per fornire un riferimento ad un nodo nello stesso workspace o in altri. Il valore della proprietà è l'UUID del nodo a cui si riferisce. Di conseguenza, solo un nodo 'referenceable' può essere di tipo riferimento. Le proprietà REFERENCE hanno la caratteristica aggiuntiva di mantentere l'integrità referenziale, impedendo la rimozione di ogni nodo che è riferito. Per rimuovere un nodo riferito è necessario prima di tutto rimuovere il riferimento. Il controllo di integrità referenziale viene effettuato quando si cerca di eseguire un'operazione di rimozione o nel momento del salvataggio delle modifiche. Il metodo

Node.getReferences()

si utilizza per trovare tutte le proprietà REFERENCE che riferiscono ad un particolare nodo. Il metodo

Node.setProperty(String name, Node value)

può essere utilizzato per impostare il valore di una proprietà avente UUID di uno specifico nodo.

2.7.3 Valori non nulli

Ogni proprietà deve avere un valore. Non si ha la possibilità di dichiarare una proprietà null o con nessun valore. Impostare una proprietà a null è equivalente alla rimozione della proprietà. Nel caso di una proprietà multi-valore se inserito un valore null sarà come non aver inserito il valore. Sarà possibile in questo caso avere una proprietà con zero valori, ad esempio se si ha una proprietà di riferimento con tre nodi riferiti, si vogliono eliminare i tre nodi, prima si dovranno eliminare i riferimenti e quindi la proprietà di riferimento non avrà più valori.

2.8 Tipi di nodi

Ogni nodo deve avere uno e soltanto un tipo di nodo primario. Il tipo di nodo primario definisce i nomi, i tipi e le caratteristiche delle proprietà e dei nodi figli che un nodo può avere. Ogni nodo ha una proprietà speciale chiamata *jcr:PrimaryType* che registra il nome del tipo di nodo primario.

In aggiunta un nodo può anche avere uno o più tipi di nodo mixin. Un nodo definito mixin ha delle ulteriori proprietà, come versioni, riferimenti. Quando ad un nodo viene assegnato il tipo di nodo mixin, acquista una particolare proprietà chiamata *jcr:MixinTypes* che registra questi nodi mixin.

Il Livello 1 delle specifiche, fornisce i metodi per scoprire i tipi di nodi già esistenti e per scoprire e leggere le definizioni dei tipi di nodi disponibili nel repository.

Il Livello 2 fornisce i metodi per l'assegnazione dei tipi di nodi primari o mixin.

La specifica non tenta di fornire metodi per la definizione, la creazione o la gestione di tipi di nodi primari o mixin, per poter definire nuovi tipi di nodo sarà necessario scrivere dei piccoli file in CND (Custom Node Type).

2.8.1 Nodi referenceable

Il concetto di nodo referenceable è fondamentale per molte funzioni del repository, comprese le versioni e i workspace multipli. I seguenti principi definiscono le caratteristiche e il funzionamento dei nodi referenceable: un repository supporta i nodi referenceable se è possibile creare al suo interno il tipo mixin *mix:referenceable*. Una volta definito un nodo referenceable è necessario assegnare alla proprietà di riferimento il valore inserito nella proprietà *jcr:uuid* identificativo univoco del

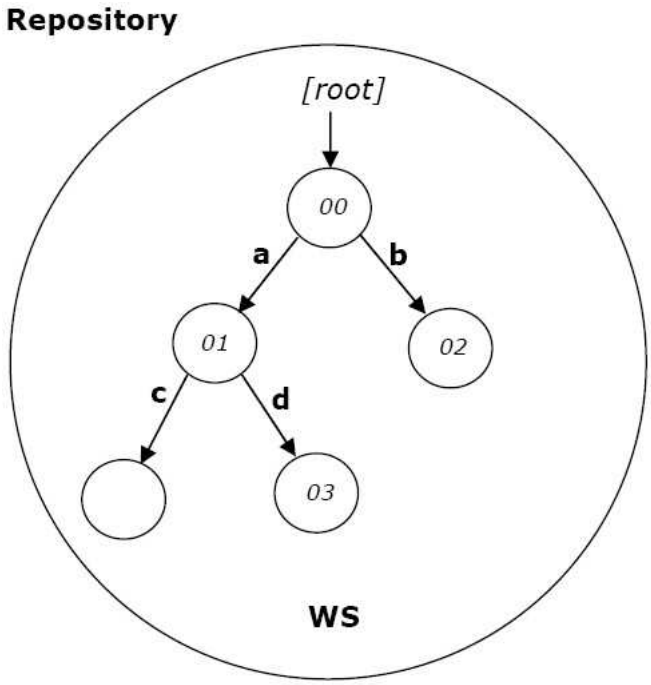

Figura 2.6: Singolo Workspace.

nodo. La proprietà jcr:uuid è protetta, creata automaticamente e obbligatoria. Questo significa che è creata e amministrata dal sistema e può soltanto essere letta dal cliente. Il lavoro della proprietà jcr:uuid è quello di esporre il nodo ad essere riconosciuto univocamente. In un dato workspace non ci sarà mai più di un nodo con uno stesso uuid.

2.8.2 Quando vengono assegnati gli UUID

L'assegnazione dell'indentificativo univoco al nodo (uuid) viene effettuata nel momento in cui si salva il nodo creato.

2.9 Workspace

Un Content Repository è composto da un numero arbitrario di aree di lavoro, workspace. Ogni workspace contiene un singolo albero di elementi. Nel caso più semplice un archivio sarà composto da un unico workspace, al contrario in casi più complessi da più workspace.

2.9.1 Repository con un unico workspace

Un repository con un unico workspace consiste di un singolo albero di nodi e proprietà. In questo caso tutti i nodi contenuti nel singolo workspace saranno identificati univocamente, nessun altro nodo avrà lo stesso uuid. La Figura 2.6 illustra un repository con un singolo workspace.

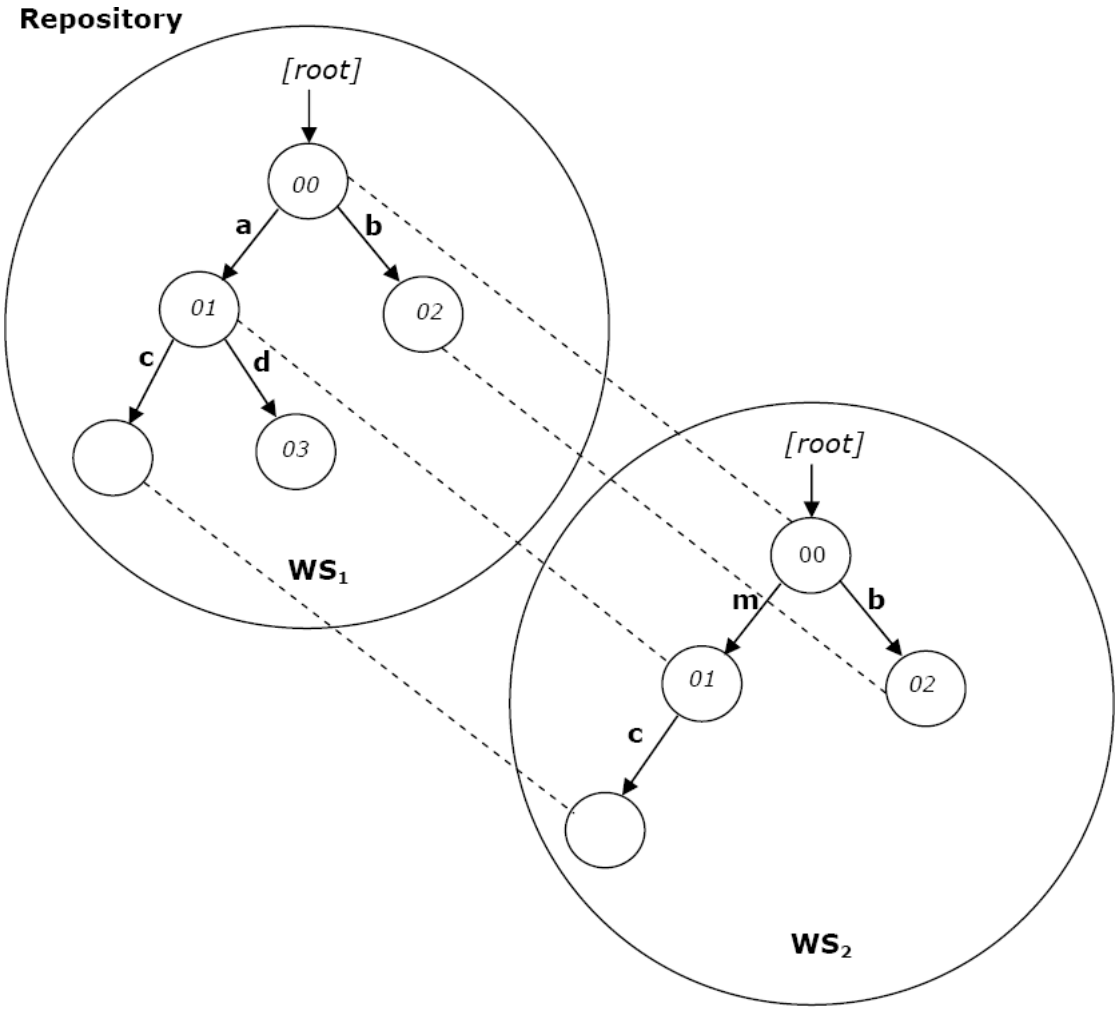

Figura 2.7: Workspace multipli.

I cerchi rappresentano i nodi, le frecce da genitore a figlio sono etichettate con il nome del figlio e identificano un riferimento. Il nome della radice è una stringa vuota, in questo caso viene indicata con root. I numeri all'interno dei nodi rappresentano l'uuid del nodo.

2.9.2 Workspace multipli e corrispondenza fra nodi

Nei repository che hanno più workspace, un nodo di un workspace può essere messo in corrispondenza con un nodo di un altro workspace. Mettere in corrispondenza significa legare due nodi, uno appartenente ad un workspace e uno di un altro workspace come la Figura 2.7 mostra. In realtà la corrispondenza è un riferimento mediante l'identificatore univoco.

2.10 Versioni

Il supporto per il controllo delle versioni è operazione opzionale. Il sistema di controllo delle versioni è stato costruito sopra al sistema dei workspace e dei nodi referenceable precedentemente descritti. In un repository che supporta il controllo delle versioni, il workspace può contenere sia nodi versionable che non versionable. Un nodo è versionable se e solo se gli è stato assegnato il tipo di nodo mixin quale *mix:versionable*, altrimenti è un nodo non versionable. I repository che non supportano il controllo delle versioni, semplicemente non forniscono questo tipo di nodo, mentre i repository che sostengono le versioni lo devono fornire. Il tipo mix:versionable è un sottotipo di mix:referenceable, quindi se un nodo è versionable è automaticamente anche referenceable.

Essere versionable significa che in qualsiasi momento il nodo può essere recuperato per alcune modifiche. La possibilità di effettuare modifiche nel tempo viene chiamata revisione o meglio versione. Il nodo si trova in uno stato di check-in. Le versioni vengono immagazzinate nello storico delle versioni in ordine di modifica. Un nodo versionable avrà quindi uno storico con le versioni immagazzinate in singoli nodi, relazionati fra loro mediante nodi predecessori e nodi successori. I nodi contententi le versioni e lo storico sono immagazzinati in un particolare archivio presente in ogni workspace

/jcr:system/jcr:versionStorage

2.10.1 Relazione tra nodi e storico delle versioni

Il rapporto fra versioni e storico delle versioni avviene mediante uuid. La storia delle versioni comprende tutte le versioni di un dato nodo versionable, questi sono immagazzinati nello stesso 'versionStorage' riservato alle versioni. All'interno di un workspace è possibile avere nodi non versionable che ovviamente non hanno uno storico. Quando viene creato un nodo mix:versionable, viene automaticamente creato uno storico delle versioni e immagazzinata la prima versione, la quale immagazzina il contenuto del nodo corrente. Un nodo versionable può essere clonato in un altro workspace, mantiene lo stesso uuid e il nuovo nodo creato mantiene associata la cronologia delle versioni esistenti. E' da notare che se un nodo è versionable è automaticamente referenceable, non c'è la necessità di dichiarare un nodo versionable anche referenceable.

Nella Figura 2.8 viene illustrato un repository che supporta il controllo delle versioni e contiene due workspace. L'archivio delle versioni è rappresentato in basso. Questo contiene lo storico delle versioni di un nodo definito versionable. I nodi bianchi non hanno versioni. Tutti i nodi versionable sono refrenceable, anche se non tutti i nodi refrenceable sono versionable. Sono presenti nei due workspace anche nodi non referenceable. Lo storico delle versioni viene rappresentato con dei cerchi sovrapposti di diverse tonalità, ogni tonalità è una revisione del nodo che può essere fatta indifferentemente da uno dei due workspace e la nuova versione verrà associata alla cronologia esistente delle versioni.

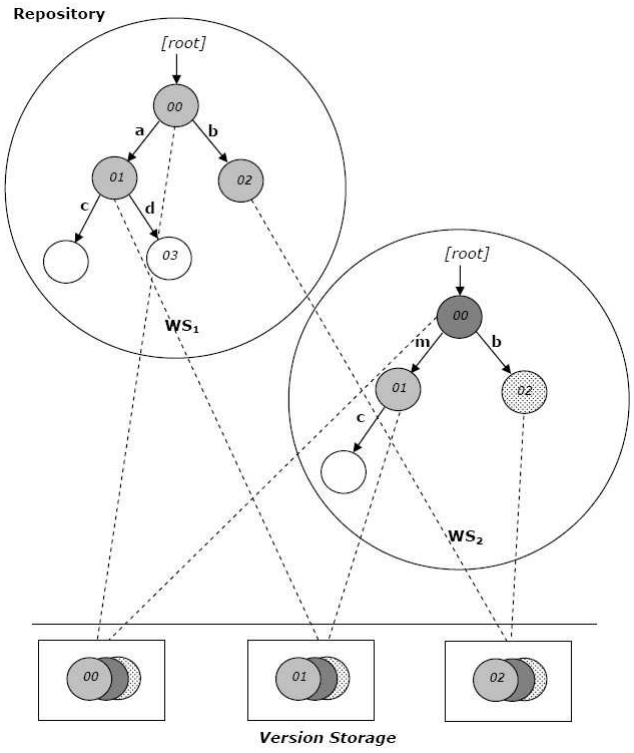

Figura 2.8: Versioni.

2.11 Metadati

L'API non fa distinzione fra contenuto reale e metadati. Tale separazione avrebbe soltanto duplicato tutte le funzionalità dell'intera API in quanto si sarebbero fornite le stesse funzionalità per la gestione del contenuto reale e dei metadati. La distinzione, in ogni caso, è significativa solo a livello di applicazione, non per il repository. Qualsiasi applicazione costruita sopra al repository può scegliere il contenuto da considerarsi metadato o contenuto reale [15].

Capitolo 3

Caratteristiche del Repository

In questo capitolo verranno trattate le specifiche JSR-170 in modo dettagliato, partendo con ordine dal Livello 1, analizzando poi il Livello 2 ed infine i Blocchi Funzionali. Saranno descritte tutte le funzionalità, mettendo in evidenza le interfacce e i metodi forniti dalle specifiche JCR per ciascun livello.

3.1 A Livello 1

Questa sezione spiega in modo funzionale il Livello 1. Esso definisce soltanto la lettura nel repository. Quindi verrà preso in considerazione un repository che implementa soltanto le caratteristiche del Livello 1.

3.1.1 Accedere al repository

Per poter accedere ad un repository è necessario per prima cosa definire un oggetto Repository che identifica il repository stesso che verrà utilizzato. Questa è la prima azione da fare, perché il punto di accesso è l'oggetto Repository. Quindi un repository verrà definito in uno dei seguenti modi, in base all'implementazione che si desidera effettuare:

Repository repository = new TransientRepository();

Repository repository = new URLRemoteRepository(URLAddress);

Repository repository = new RMIRemoteRepository(RMIAddress);

Repository repository = new JNDIRemoteRepository(URLAddress);

Nel primo caso il repository sarà locale, negli altri tre si avrà il repository nel server e un client che interagisce con quest'ultimo, è possibile che la stessa macchina svolga il ruolo sia del server che del client oppure che entrambi stiano in due macchine differenti.

Qualsiasi tipo di meccanismo usato per dichiarare un Repository restituirà un oggetto che implementa l'interfaccia Repository contenuta nel pacchetto javax.jcr.

Fatto ciò, per entrare nel repository, sarà necessario effettuare un login per ottenere una sessione. I possibili metodi per loggarsi ad un repository sono:

- **login(Credentials credentials, String workspaceName)**: Autentica l'utente utilizzando le credenziali fornite. Se il Workspace passato nella firma del metodo è riconosciuto come il nome di un workspace valido per il repository e l'autorizzazione per accedere a tale area è concessa, allora viene ritornato un nuovo oggetto Session. Se le credenziali sono nulle, si presume che l'autenticazione sia gestita da un meccanismo esterno al repository stesso (ad esempio, attraverso le JAAS) e che l'implementazione del repository esista all'interno di un determinato contesto (per esempio, un application server) che permette di gestire l'autorizzazione di accesso allo spazio di lavoro specificato, in questo caso l'accesso non è gestito. Se il nome del workspace è nullo, verrà selezionata automaticamente un'area di lavoro, in genere il workspace di default. E' possibile il lancio delle eccezioni: *LoginException* nel caso di credenziali scorrette, *NoSuchWorkspaceException* se il nome del workspace non viene riconosciuto.

- **login(Credentials credentials)**: Equivalente a login(credentials, null).

- **login(String workspaceName)**: Equivalente a login(null, workspaceName).

- **login()**: Equivalente a login(null, null).

3.1.2 Descrittori del repository

I metodi

Repository.getDescriptorKeys
e
Repository.getDescriptorKey(String)

possono essere utilizzati per ricavare informazioni circa una particolare implementazione del repository.

I descrittori sono delle Stringhe costanti che possono assumere due valori, *true* se il repository implementa la funzione descritta, *false* se non implementa una data funzione. La Tabella 3.1 presenta alcuni dei descrittori; si può vedere che ciascuna funzione che il repository può implementare avrà un descrittore associato, con il relativo valore.

3.1.3 Credenziali

Nel momento in cui si effettua un login servono delle credenziali per identificare l'utente. Le credenziali usate per l'autenticazione devono implementare l'interfaccia Credentials. E' possibile realizzare un'applicazione personalizzata o avvalersi di SimpleCredentials presente nel pacchetto javax.jcr. Essa fornisce dei metodi standard minimi per l'autenticazione di un utente in un repository, utilizzando username e password. I metodi vengono messi in evidenza in seguito:

Tabella 3.1: Esempio di alcuni descrittori

Descriptor Key (Stringhe costanti)	**Informazioni ritornate**
SPEC_VERSION_DESC	Indica la versione delle specifiche jcr in questo caso '1.0'.
SPEC_NAME_DESC	Indica il nome delle specifiche, ovvero 'Content Repository for Java Technology API'.
REP_NAME_DESC	Il nome del repository.
REP_VERSION_DESC	La versione del repository.
LEVEL_1_SUPPORTED	Indica se il repository supporta le funzionalità del Livello 1 .
LEVEL_2_SUPPORTED	Indica se il repository supporta le funzionalità del Livello 2.
OPTION_TRANSACTIONS_SUPPORTED	Indica se l'implementazione gestisce le transazioni.
OPTION_VERSIONING_SUPPORTED	Indica se vengono supportate le versioni.
...	...

- **SimpleCredentials(String userID, char[] password)**: Crea un nuovo oggetto SimpleCredentials con username e password.

- **getUserID()**: Ritorna l'username dell'utente.

- **getPassword()**: Ritorna la password dell'utente.

- **setAttribute(String name, Object value)**: Immagazzina un attributo nell'oggetto Credentials.

- **removeAttribute(String name)**: Rimuove un attributo dall'oggetto Credentials.

- **getAttribute(String name)**: Ritorna l'oggetto associato all'attributo con il nome richiesto.

- **getAttributeNames()**: Ritorna i nomi di tutti gli attributi disponibili nell'oggetto Credentials.

3.2 Leggere il contenuto del repository

Leggere il contenuto del repository significa accedere ai nodi direttamente o indirettamente attraversando ciascun nodo passo per passo e leggere i rispettivi valori delle proprietà. L'oggetto Session restituito da

Repository.login

comprende sia le autorizzazioni di un particolare utente sia il legame con un workspace, entrambi parametri passati al momento del login. Ogni oggetto Session è associato ad un oggetto Workspace. L'oggetto Workspace rappresenta un'entità effettiva di lavoro presente nel repository. Vi è una distinzione importante tra l'istanza dell'oggetto di lavoro associato ad una particolare sessione e l'entità effettiva di lavoro nel repository. Se l'accesso ad un particolare workspace avviene da parte di tante sessioni, ciascuna avrà il proprio spazio di lavoro, anche se tutti questi oggetti possono rappresentare la stessa entità di lavoro effettiva nel repository. In altre parole, un oggetto di lavoro corrisponde ad una visione di una particolare entità di lavoro e la vista è determinata dalla sessione associata all'oggetto Workspace. Ogni oggetto sessione rappresenta un'entità separata. Dal momento che la sessione e il workspace sembrano combinati in un unico oggetto, questa illusione sarà soltanto presente nella lettura del repository a Livello 1. La distinzione fra i due oggetti entra in gioco a Livello 2 nel quale la scrittura può avvenire in due modi, sia attraverso la memorizzazione temporanea (Session) o direttamente a livello di archiviazione (Workspace).

La distinzione tra gli oggetti session e workspace a Livello 1 non è rilevante, esiste per motivi di compatibilità con il Livello 2.

3.2.1 Metodi della Session

Di seguito sono riportati alcuni dei metodi disponibili, utilizzati per leggere nel repository e accedere ai contenuti del workspace in lettura. I metodi più importanti della Session sono quelli che forniscono l'accesso agli elementi del workspace: in genere l'utente come primo passo invoca

Session.getRootNode()

il quale restituisce il nodo radice. Da questo nodo l'utente può attraversare l'albero di nodi presente nel workspace. E' anche possibile accedere direttamente ad un nodo nello spazio di lavoro con

Session.getNodeByUUID()
o
Session.getItem()

dove è già noto che Item può rappresentare un elemento, quale nodo o proprietà.

javax.jcr.Session fornisce metodi per l'analisi del repository come:

- **getRepository()**: Restituisce l'oggetto Repository dal quale è stato eseguito un login per ottenere la data Session.

- **getUserID()**: Si ottiene il nome dell'utente della sessione, che era stato assegnato al momento del login con le credenziali. E' possibile fare un login con un utente anonimo o null.

- **getAttributeNames()**: Restituisce i nomi degli attributi impostati in questa sessione, se non sono stati assegnati attributi alle credenziali della sessione, verrà restituito l'array vuoto.

- **getAttribute(String name)**: Restituisce il valore del nome dell'attributo passato come parametro di input al metodo, o null se non ci sono attributi con quel dato nome.

- **getWorkspace()**: Ritorna il workspace assegnato a quella sessione.

- **itemExists(String absPath)**: Ritorna true se esiste un elemento avente l'assoluto path specificato nel parametro di input, altrimenti false. Se si verificano errori verrà lanciata un'eccezione *RepositoryException*.

- **impersonate(Credentials c)**: Ritorna una nuova sessione con le nuove credenziali specificate. Consente quindi all'utente corrente di 'rappresentare' un altro utente assumendo ovviamente che la nuova sessione dia dei permessi o delle restrizioni. Viene lanciata un'eccezione *LoginException* se la sessione non dispone di autorizzazioni sufficienti per compiere l'operazione.

- **logout()**: Rilascia tutte le risorse associate alla sessione. Questo metodo dovrebbe essere sempre invocato quando si desidera uscire dalla sessione.

- **isLive()**: Ritorna true se l'oggetto è utilizzabile dall'utente.

3.2.2 Metodi di lettura del Workspace

A Livello 1 l'oggetto Workspace serve soltanto per incapsulare una serie di metodi per l'accesso e per le informazioni ulteriori sulle funzionalità del repository. javax.jcr.Workspace fornisce:

- **getSession()**: Ritorna l'oggetto Session attraverso il quale l'oggetto Workspace è stato acquisito.

- **getName()**: Restituisce il nome dello spazio di lavoro effettivo rappresentato dall'oggetto Workspace. Questo è il nome usato da *Repository.login*.

- **getQueryManager()**: Restituisce il QueryManager, attraverso il quale i metodi di ricerca sono accessibili. Viene generata *RepositoryException* se si verifica un errore.

- **getNamespaceRegistry()**: Restituisce l'oggetto NamespaceRegistry, che può essere utilizzato per accedere al registro dei prefissi e dei nomi dei nodi.

- **getNodeTypeManager()**: Restituisce il NodeTypeManager utilizzato per accedere a informazioni sui tipi di nodo disponibili nel repository. Quindi il NodeTypeManager non è un'area di lavoro specifico, ma fornisce i metodi per l'introspezione globale dei tipi di nodi disponibili.

- **getAccessibleWorkspaceNames()**: Ritorna un array di stringhe contenente i nomi di tutte le aree di lavoro in questo repository, accessibili dall'utente avente le credenziali utilizzate per ottenere la sessione.

3.2.3 Metodi per leggere i nodi

In seguito sono presentati i metodi a Livello 1 supportati per la lettura dei nodi. Questi vengono utilizzati per ottenere tutti i nodi e le rispettive proprietà. javax.jcr.Node fornisce i metodi per analizzare i nodi presenti nel repository:

- **getNode(String relPath)**: Restituisce il nodo avente path relativo assegnato in input. Se il path è comune a più nodi verranno ritornati tutti i nodi con il path richiesto, contraddistinti da un numero d'ordine inserito all'interno di parentesi quadre. Se il nodo, al path richiesto, non esiste, verrà lanciata un'eccezione del tipo *PathNotFoundException*.

- **getNodes()**: Ritorna tutti i nodi figli del nodo, indicati soltanto dal nome, non vengono mostrate le proprietà. Se il nodo non ha figli viene restituito un iteratore vuoto.

- **getProperty(String relPath)**: Invocando questo metodo viene ritornata la proprietà indicata in input. Se questa non esiste verrà lanciata un'eccezione di tipo *PathNotFoundException*.

- **getProperties()**: Si ottengono tutte le proprietà del nodo. Se il nodo non ha alcuna proprietà viene ritornato un iteratore vuoto.

- **getPrimaryItem()**: Viene ritornato il tipo di nodo primario associato al nodo, se esiste.

- **getUUID()**: Ritorna l'UUID del nodo.

- **getIndex()**: Ritorna l'indice di 'omonimia' fra i nodi. Per i nodi che non hanno nomi uguali ad altri nodi verrà restituito il valore 1.

- **getReferences()**: Restituisce tutte le proprietà REFERENCE che fanno riferimento a questo nodo. Se non ci sono riferimenti verrà restituito un iteratore vuoto.

- **hasNode(String relPath)**: Restituisce true se esiste il nodo rischiesto al determinato percorso, false altrimenti.

- **hasNodes()**: Restituisce true se il nodo ha uno o più nodi figli, false altrimenti.

- **hasProperty(String relPath)**: Restituisce true se esiste la proprietà con il nome dato e false altrimenti.

- **hasProperties()**: Restituisce true se il nodo ha uno o più proprietà, false altrimenti.

3.2.4 Metodi di lettura delle proprietà

In javax.jcr.Property sono contenuti metodi che permettono di leggere le proprietà di un nodo, e sono:

- **getValue()**: Restituisce il valore della proprietà mediante un oggetto di tipo Value. Se questa proprietà è multivalore, il metodo genera *ValueFormatException*. L'oggetto restituito è una copia del valore memorizzato ed è immutabile.

- **getValues()**: Restituisce un array di tutti i valori della proprietà. Questo metodo viene utilizzato nelle proprietà multivalore. Se la proprietà ha un singolo valore, viene generata l'eccezione *ValueFormatException*. L'array restituito è una copia dei valori memorizzati, non si possono effettuare cambiamenti.

- **getString()**: Restituisce il valore della proprietà sottoforma di oggetto String. Se la proprietà contiene più valori oppure non è possibile convertirla in String viene generata un'eccezione *ValueFormatException*.

- **getStream()**: Ritorna un InputStream: rappresentazione del valore della proprietà. Se la proprietà contiene più valori viene generata un'eccezione *ValueFormatException*.

- **getLong()**: Restituisce il valore della proprietà mediante l'oggetto intero lungo, Long. Se la proprietà contiene più valori oppure non è possibile convertirla in Long viene generata un'eccezione *ValueFormatException*.

- **getDouble()**: Ritorna il valore della proprietà sottoforma di Double. Se la proprietà contiene più valori oppure non è possibile convertirla in Double viene generata l'eccezione *ValueFormatException*.

- **getDate()**: Si ottiene il valore della proprietà mediante l'oggetto Calendar. Se la proprietà contiene più valori oppure non è possibile convertirla in un oggetto Calendar viene generata un'eccezione *ValueFormatException*.

- **getBoolean()**: Restituisce il valore della proprietà con un oggetto Boolean. Se la proprietà contiene più valori oppure non è possibile convertirla in Boolean viene generata l'eccezione *ValueFormatException*.

- **getNode()**: Si ottiene con questo metodo il valore di una proprietà che immagazzina dei nodi. Ne è di esempio la proprietà del tipo REFERENCE,

dove il valore di tale proprietà è l'UUID del nodo, quindi il nodo stesso. Se la proprietà è multivalore o non può essere convertita in un riferimento, in quanto non è un nodo, questo metodo genera *ValueFormatException*.

- **getLength()**: Restituisce la lunghezza del valore di questa proprietà in byte se il valore è un PropertyType.BINARY, altrimenti restituisce il numero di caratteri necessari per visualizzare il valore nella sua forma di stringa. Restituisce -1 se non è possibile determinare la lunghezza del valore. Se questa proprietà è multivalore, questo metodo genera *ValueFormatException*.

- **getType()**: Ritorna il tipo della proprietà. Il tipo restituito è quello fissato al momento della creazione della proprietà. Nella prossima sezione saranno presentati i possibili tipi di proprietà.

3.2.5 Tipi di proprietà

La classe PropertyType definisce delle costanti di tipo Integer che identificano i tipi di proprietà disponibili e i nomi standard assegnati. Fornisce, inoltre, due metodi per la conversione tra il nome della proprietà e il valore intero e viceversa. Le proprietà possono assumere uno dei tipi illustrati in seguito:

- **STRING**: Il tipo di proprietà String viene usato per archiviare le stringhe. Ha le stesse caratteristiche della classe Java String.

- **BINARY**: Le proprietà di tipo Binary vengono usate per memorizzare dati binari.

- **LONG**: Il tipo di proprietà Long viene utilizzato per memorizzare numeri interi lunghi. Ha le stesse caratteristiche del tipo primitivo Java Long.

- **DOUBLE**: Il tipo di proprietà Double si usa per memorizzare i numeri in virgola mobile. Ha le stesse caratteristiche del tipo primitivo Java Double.

- **BOOLEAN**: Il tipo di proprietà Boolean è utilizzato per memorizzare i valori boolean. Ha le stesse caratteristiche del tipo primitivo Java Boolean.

- **DATE**: Il tipo data è utilizzato per memorizzare dati nel formato data-ora.

- **NAME**: Il nome è una associazione di uno spazio (namespace) e di un nome locale (local name). Quando viene letto, il namespace è mappato al prefisso corrente.

- **PATH**: Una proprietà di tipo path identifica un percorso. Quest'ultimo può essere un percorso relativo o assoluto.

- **REFERENCE**: Una proprietà di tipo riferimento immagazzina l'identificatore univoco (UUID) del nodo al quale riferisce. La proprietà di riferimento garantisce integrità referenziale.

- **UNDEFINED**: Questa costante può essere utilizzata all'interno della definizione di una proprietà, per specificare che la proprietà in questione potrà essere di qualsiasi tipo.

- **nameFromValue(int type)**: Ritorna il nome standard di un dato tipo di proprietà, specificato dal valore intero passato nella firma del motodo.

- **valueFromName(String name)**: Restituisce il valore intero del dato tipo di proprietà, avente il nome standard specificato nella firma del metodo.

Date Il formato della data rispetta l'ISO 8601:2000 e verrà visualizzata nel seguente modo: sYYYY-MM-DDThh:mm:ss.sssTZD. Dove sYYYY indica l'anno, se s è negativo si tratterà di un anno a. C. altrimenti se positivo d. C.. MM indica il mese, DD il giorno, hh l'ora, mm i minuti, ss.sss i secondi con tre cifre decimali. Le tre lettere TZD rappresentano il Time Zone Designator ovvero il fuso orario.

Name Un NAME è un'associazione di un namespace e di un local name. Questa associazione deve essere gestita internamente in modo tale che nel momento in cui viene letta attraverso le specifiche, il namespace è mappato al corrente prefisso. Ad esempio, se al momento della lettura il prefisso corrente all'URI mappato è:

$$myapp \text{ -> } http://mycorp.com/myapp$$

allora il name pienamente qualificato sarà:

$$\{http://mycorp.com/myapp\}myItem$$

che verrà ritornato come stringa in questo modo:

$$myapp{:}myItem$$

Se il namespace in futuro verrà rimappato, ad esempio in:

$$yourapp \text{ -> } http://mycorp.com/myapp$$

allora il valore ritornato cambierà e sarà:

$$yourapp{:}myItem$$

La proprietà di tipo name viene utilizzata per la registrazione dei nuovi tipi di nodo.

Path Una proprietà di tipo PATH è una lista ordinata di elementi. Un elemento che identifica un percorso è un nome con un indice opzionale. Al momento della lettura, i nomi completi vengono mappati nel percorso. Un percorso può essere assoluto o relativo. Ad esempio, data la mappatura dello spazio dei nomi:

myapp -> http://mycorp.com/myapp

allora il PATH pienamente qualificato sarà:

/{http://mycorp.com/myapp}document[1]/
{http://mycorp.com/myapp}paragraph[3]

Ovviamente verrà ritornato, quindi sarà visibile all'utente, nel modo seguente:

/myapp:document/myapp:paragraph[3]

Se il namespace viene rimappato nel tempo anche i rispettivi path subiranno le modifiche.

Un uso comune delle proprietà PATH si ha quando si vogliono conoscere i percorsi degli elementi immagazzinati nel workspace; quanto specificato in precedenza è visibile nel momento in cui si desidera visualizzare lo storico delle versioni. Il repository non applica l'integrità referenziale alle proprietà di tipo PATH.

Reference Una proprietà di tipo REFERENCE immagazzina l'UUID del nodo al quale riferisce. L'integrità referenziale deve essere ed è garantita, ne è un esempio la rimozione di un nodo che contiene riferimenti nelle applicazioni gestite a Livello 2. Per esso l'integrità è garantita, cioè se si vuole eliminare tale nodo non sarà possibile, prima sarà necessario eliminare il riferimento ed in seguito eliminare il nodo. Se si tenta di eliminare un nodo con dei riferimenti verrà generata l'eccezione *ReferentialIntegrityException*.

3.3 Namespace

Un repository di contenuti fornisce il supporto per il namespace degli elementi e per i nomi dei tipi di nodo. Il namespace serve a prevenire le collisioni di denominazione tra elementi e tipi di nodi che provengono da fonti diverse o domini di applicazione.

3.3.1 Registro del Namespace

Ogni archivio ha un unico registro per immagazzinare gli spazi dei nomi, i namespace, questo è rappresentato dall'oggetto NamespaceRegistry, al quale si accede tramite

Workspace.getNamespaceRegistry()

I metodi contenuti in javax.jcr.NamespaceRegistry supportati a Livello 1 sono:

- **getPrefixes()**: Ritorna un array di tutti i prefissi registrati.
- **getURIs()**: Ritorna un array di tutti gli URIs registrati.
- **getURI(String prefix)**: Ritorna l'URI di un dato prefisso mappato. Se lo specifico prefisso non esiste verrà lanciata un'eccezione di tipo *NamespaceException*.
- **getPrefix(String uri)**: Ritorna il prefisso mappato associato al determinato URI, se non esiste viene lanciata un'eccezione *NamespaceException*.

NamespaceRegistry ha anche altri metodi utilizzabili soltanto a Livello 2 come l'aggiunta o la rimozione di namespace.

Un prefisso registrato può essere usato in un nome di un nodo o di una proprietà all'interno del repository. Il prefisso rappresenta una scorciatoia per l'URI associato. Si ha quindi un'unicità universale per ciascun namespace.

All'interno del repository sono sempre contenuti i seguenti namespace:

- *jcr -> http://www.jcp.org/jcr/1.0*: Riservato per gli elementi definiti all'interno per la costruzione dei tipi di nodi. Ad esempio jcr:content.
- *nt -> http://www.jcp.org/jcr/nt/1.0*: Riservato per i nomi dei tipi di nodi primari. Ad esempio nt:file.
- *mix -> http://www.jcp.org/jcr/mix/1.0*: Riservato per i nomi dei tipi di nodi mixin. Ad esempio mix:referenceable.
- *xml -> http://www.w3.org/XML/1998/namespace*: Riservato per ragioni di compatibilità con il linguaggio XML. Questo prefisso non deve mai essere usato, perchè potrebbe causare problemi nel momento in cui si desidera effettuare l'esportazione in formato XML.

A Livello 1 non è possibile aggiungere nuovi namespace, ma se necessario, è possibile rimappare un namespace esistente.

3.3.2 Rimappare in una data sessione il Namespace

Qualsiasi namespace precedentemente registrato in una sessione può essere temporaneamente rimappato, in questo modo si avrà un nuovo prefisso al posto di quello esistente.

Un caso d'uso che si basa sul namespace si verifica nel contesto di query XPath o SQL. Le query spesso includono dei nomi letterali che sono prefissi di namespace. Quando si tenta di utilizzare una query memorizzata (nt:query) o ottenuta da una fonte esterna, dove i prefissi non corrispondono a quelli attualmente utilizzati nel repository, la rimappatura dinamica dei namespace nel corso della sessione consente di adattare i prefissi per poter ottenere, come in questo esempio, il risultato della query.

Quando viene rimappato un namespace, lo spazio dei nomi assegnato in maniera temporanea può essere utilizzato da tutti gli oggetti presenti nella sessione dove è stata effettuata la rimappatura.

3.3.3 Immagazzinare internamente i nomi e i percorsi

Si noti che i nomi dei nodi e delle proprietà devono essere memorizzati internamente in modo tale che quando si accede ad essi, l'accesso riflette l'attuale mappatura del namespace. Un modo per raggiungere tale obiettivo è quello di memorizzare internamente utilizzando nomi qualificati e, al momento dell'accesso, produrre dinamicamente il corretto prefisso o il percorso basato sulla mappatura corrente.

3.4 Esportare contenuti del Content Repository

Il Livello 1 supporta l'esportazione dei contenuti del repository mediante XML. L'XML può essere trasmesso sia come uno stream di dati o mediante lo standard SAX utilizzato per memorizzare un file xml. I flussi XML prodotti dall'esportazione devono essere codificati in UTF-8. I motodi per le esportazioni si possono ottenere dall'oggetto Session.

3.5 Ricercare all'interno del repository

A livello 1 la sintassi XPath è obbligatoria. Opzionalmente, un repository può sostenere anche il linguaggio SQL. XPath è un linguaggio di ricerca originariamente progettato per la selezione di elementi da un documento XML. Dal momento che un workspace, come un documento XML, può essere visto come una struttura ad albero, XPath fornisce una sintassi conveniente per la ricerca dei contenuti all'interno del repository.

3.5.1 Struttura di una query

Una query, sia XPath che SQL, specifica un sottoinsieme di nodi all'interno di un workspace, chiamato 'set di risultati' della query. Il set di risultati rappresenta tutti i nodi del workspace che soddisfano i vincoli indicati nella query. Il risultato della query viene restituito in due forme parallele: un iteratore per i nodi e una 'tabella' in cui ogni riga contiene tutte le proprietà del nodo. Nella Tabella 3.2 verranno presentate delle query di base in linguaggio XPath.

3.5.2 Le estensioni del linguaggio XPath

XPath nel Content Repository definisce anche un limitato numero di funzioni in aggiunta a quelle presentate nella Tabella 3.2. Queste funzioni hanno come prefisso *jcr:*.

jcr:like Tale funzione è basata sul predicato LIKE come nel linguaggio SQL. Ad esempio, per trovare tutti i nodi che hanno come valore della proprietà title la sottostringa 'Java', la sintassi sarà la seguente

Tabella 3.2: Esempi Query XPath

Esempi Query XPath

*//**	I caratteri // indicano che i nodi potranno trovarsi in qualsiasi percorso e * pone che sia ritornato un nodo avente tipo di nodo indifferente. Questa query seleziona tutti i nodi contenuti all'interno del workspace.
//element(,my:type)*	Seleziona su un qualsiasi percorso tutti i nodi di un determinato tipo, in questo caso my:type. Il tipo di nodo my:type rappresenta un nodo personalizzato che verrà analizzato a Livello 2.
//element(,my:type) /@my:title*	Seleziona su un qualsiasi percorso i nodi di un determinato tipo aventi una determinata proprietà che sarà restituita nei risultati, nel caso in esame my:title. Da notare che le proprietà devono essere precedute dal simbolo @.
//element(,my:type)/ (@my:title | @my:text)*	Come nel caso precedente soltanto che la ricerca includerà due proprietà.
/jcr:root/nodes// element(,my:type)*	Seleziona tutti i nodi di un determinato tipo, in questo caso my:type con la restrizione di ricerca in un determinato percorso.
//element(,my:type) [@my:title = 'JSR 170']*	Seleziona su un qualsiasi percorso i nodi di un determinato tipo aventi una determinata proprietà con il rispettivo valore richiesto. Vengono utilizzati gli operatori di uguaglianza, minoranza e maggioranza come nella sintassi SQL.
//element(,my:type) order by @my:title*	Seleziona su un qualsiasi percorso tutti i nodi di un determinato tipo ordinandoli in base alla proprietà.
//element(,my:type) order by @my:date descending, @my:title ascending*	Seleziona su un qualsiasi percorso tutti i nodi di un determinato tipo ordinandoli in base alla proprietà decidendo se in ordine crescente o decrescente.
//element(,my:type) [jcr:contains(.,'jcr')]*	Questa sarà una delle query maggiormente utilizzate, permette di ricercare nel contenuto di tutti i nodi di tipo my:type la parola assegnata fra virgolette, nel caso proposto jcr.

//[jcr:like(@title, '% Java %')]

Come nel linguaggio SQL il % rappresenta una stringa di zero o più caratteri e il simbolo _ un singolo carattere.

jcr:contains Questa funzione è utilizzata per le ricerche full text.

//[jcr:contains(., 'contenuto da cercare')]

Il primo parametro, fra le parentesi tonde, specifica il campo di applicazione solitamente rappresentato da '.', in esso vengono impersonificate sia il contenuto del documento che le proprietà. Il secondo paramentro rappresenta il testo da cercare nel contenuto del documento, il contenuto risulta tutto indicizzato automaticamente. Ovviamente *jcr:contains* è supportato se il contenuto del nodo è stato indicizzato. Questo è quanto viene effettuato anche dai motori di ricerca come Google.

jcr:deref La funzione *jcr:deref* permette di trovare le proprietà di tipo REFERENCE all'interno del workspace. Il supporto di questa funzione è opzionale. La sintassi da utilizzare è:

//jcr:deref(@my:author, 'my:person')

dove nel primo parametro verrà inserito il nome della proprietà di riferimento e nel secondo dovrà essere passato il tipo di nodo che supporta le proprietà di tipo reference. L'espressione mostrata restituirà tutti i nodi di tipo my:person che contengono la proprietà di riferimento my:author.

3.5.3 La ricerca: Query API

Le query di ricerca analizzano i dati già immagazzinati nel repository, non lavorano a livello di dati non ancora salvati nel repository.

E' possibile accedere al contenuto del repository, mediante una ricerca, grazie all'oggetto QueryManager. Esso viene chiamato dal workspace con il metodo

getQueryManager()

da esso sarà possibile accedere a tutti i metodi per interrogare il repository.

L'oggetto QueryManager fornisce il metodo

QueryManager.createQuery(String statement, String language)

il quale permette di creare una nuova query specificando l'istruzione della query stessa e il linguaggio con cui la query verrà scritta. Se l'istruzione di query è sintatticamente non valida, dato il linguaggio specificato, un *InvalidQueryException* viene generata. Il parametro del linguaggio usato deve essere una stringa tra quelle restituite da

QueryManager.getSupportedQueryLanguages()

se così non fosse, allora un *InvalidQueryException* verrà generata.

getSupportedQueryLanguages()

Restituisce un array di stringhe che individuano i linguaggi supportati. A Livello 1, la stringa sarà rappresentata da Query.XPATH. Se è supportato addizionalmente anche il linguaggio SQL si avrà anche Query.SQL.

Nel momento in cui si invoca il metodo

QueryManager.createQuery

viene creata una nuova Query, un oggetto di tipo Query.

In questo caso si tratta di una query transitoria utilizzabile a Livello 1. Se invece si lavora a Livello 2, è supportato il tipo di nodo nt:query, quindi la query transitoria può essere memorizzata nel repository chiamando

Query.storeAsNode(absPath String)

Questo crea un nodo di tipo nt:query nel percorso specificato. Per recuperare una query memorizzata viene invocato il metodo

QueryManager.getQuery(Node node)

dove il nodo è di tipo nt:query.

Da tenere presente che se viene immagazzinata una query e in seguito rimappato un namespace, la query non produrrà più i risultati che si sarebbero ottenuti prima della rimappatura. Questo significa che è nettamente conveniente eseguire una query nel momento in cui serve.

Una volta che una query è stata definita, può essere eseguita. Il metodo

Query.execute()

restituisce un QueryResult. I risultati restituiti rispettano le limitazioni di accesso associate alla sessione corrente. In altre parole, se la sessione corrente non dispone delle autorizzazioni di lettura di un elemento particolare, tale elemento non sarà incluso nel set dei risultati. Le query vengono eseguite nel contenuto effettivamente memorizzato nel repository.

Il QueryResult può essere restituito in due formati: con una tabella formata da colonne e righe di valori oppure con una lista di nodi. I metodi seguenti provvedono ad estrarre i valori dal QueryResult:

- **getColumnNames()**: Restituisce un array di tutti i nomi delle colonne della tabella del gruppo di risultati.
- **getRows()**: Restituisce un iteratore delle righe della tabella dei risultati. Se è stata specificata la clausola ORDER BY verrà rispettato l'ordine delle righe, quindi anche l'iteratore rifletterà l'ordine. Se l'esecuzione della query non trova elementi, verrà restituito un iteratore vuoto.
- **getNodes()**: Restituisce un iteratore di tutti i nodi che corrispondono al risultato della query. Si comporta allo stesso modo del *getRows()*.

3.6 Tipi di nodi

Una caratteristica molto importante del repository è la capacità di distinguere i soggetti memorizzati nel repository per tipo. In un Content Repository, questo è eseguito tramite l'assegnazione di tipi di nodi ai nodi. Il Livello 1 supporta:

- La scoperta dei tipi di nodi primari e mixin, per nodi già presenti nel repository.
- La scoperta dei tipi di nodi supportati nel repository.
- La scoperta della definizione dei tipi di nodi supportati.
- La scoperta dei vincoli posti su un nodo esistente.

Il Livello 2 addizionalmente specifica metodi per:

- L'assegnazione di un tipo di nodo primario nella creazione di un nuovo nodo personalizzato.
- L'assegnazione opzionale dei tipi di nodi mixin.

In questa sezione verranno spiegate le funzionalità dei tipi di nodo a Livello 1. Non verranno trattati metodi di definizione, creazione o gestione dei nodi, ma ci si limiterà a scoprire i tipi di nodi.

3.6.1 Ciò che costituisce un tipo di nodo

In un repository, un tipo di nodo definisce quali nodi figli o quali proprietà un nodo può (o deve) avere. E' necessario quindi fornire un insieme di metodi di individuazione delle informazioni sul tipo di nodo. A tal fine, questa specifica stabilisce che ogni tipo di nodo ha i seguenti attributi:

- **Name:** Ogni tipo di nodo registrato nel Content Repository possiede un suo nome univoco. Le convenzioni di denominazione per i tipi di nodo sono gli stessi che per gli elementi. Tutti i tipi di nodi primari predefiniti hanno prefisso nt. I tipi di nodo mixin sono preceduti dal prefisso mix.
- **Supertypes:** Un tipo di nodo primario (con eccezione del tipo nt:base) deve estendere un altro tipo di nodo (o più di un tipo di nodo, se l'applicazione supporta l'ereditarietà multipla). Un tipo mixin non ha un supertipo quindi non estende un tipo di nodo.
- **Mixin Status:** Un tipo di nodo può essere primario o mixin. Questo stato è parte della definizione del tipo di nodo.
- **Orderable child nodes status:** Un tipo di nodo primario può specificare che i nodi figli siano client-ordinabili. Se questo è stato impostato su true, tutti i nodi di questo tipo devono supportare il metodo *Node.orderBefore*. Se è stato impostato su false, i nodi di questo tipo non possono sostenere

questo metodo. Questa impostazione su un tipo di nodo mixin non avrà alcun effetto.

- **Property definitions:** Un tipo di nodo contiene una serie di definizioni che specificano le proprietà di un nodo e le caratteristiche che i nodi di questo tipo possono avere.

- **Child node definitions:** Un tipo di nodo contiene una serie di definizioni che specificano i nodi figli che si possono avere e le loro caratteristiche.

- **Primary Item Name:** Un tipo di nodo può specificare un elemento figlio come elemento primario. Questo indicatore è utilizzato dal metodo *Node.getPrimaryItem()*.

3.6.2 Scoperta dei tipi di nodo a Livello 1

Si noti che a Livello 1 i clienti che vorranno attuare un repository non saranno in grado di riordinare, aggiungere o rimuovere i nodi o modificare le proprietà. In un Content Repository, ogni nodo ha uno ed un unico tipo di nodo primario. Questo tipo di nodo definisce, come detto, una serie di restrizioni sugli elementi figli del nodo. In aggiunta al suo unico tipo di nodo primario, un nodo può anche avere un numero qualsiasi di mixin type di nodi ad esso assegnati. Un tipo mixin è simile a un tipo primario in quanto la sua definizione ha gli stessi parametri. Si distingue, però, in quanto prevede funzionalità aggiuntive ad un nodo. Inoltre, mentre un tipo di nodo primario può essere 'istanziato' come un nodo, non è questo il caso con i tipi mixin. Un tipo mixin non può servire, di per sé, per definire la struttura di un nodo, ma aggiunge solo le proprietà ad un nodo figlio di un nodo che ha già un tipo di nodo primario definito.

3.6.3 Le proprietà speciali jcr:primaryType e jcr:mixinType

Quando un nodo viene creato nel repository deve essere memorizzato come tipo di nodo primario mediante la proprietà chiamata *jcr:primaryType* di tipo name. In maniera simile si comportano anche i nodi mixin, questi vengono memorizzati dalla proprietà *jcr:mixinType*. Da notare che un tipo di nodo mixin deve essere esplicitamente dichiarato con

Node.addMixin(mixinType)

Queste proprietà servono per mantenere le informazioni sui tipi di nodi, sono protette, non possono essere rimosse o cambiate usando le specifiche JCR che si stanno esaminado. jcr:primaryType e jcr:mixinType sono proprietà definite nel nodo primario predefinito nt:base, il quale è supertipo di tutti i nodi, sia di quelli già definiti, sia dei nodi personalizzati che verranno aggiunti.

3.6.4 Definire le proprietà di un nodo

La definizione delle proprietà contiene le seguenti informazioni:

- **Name**: Il nome della proprietà.
- **Type**: Il tipo di proprietà richiesto. C'è la possibilità che il tipo della proprietà non sia definito, quindi la proprietà può essere di qualsiasi tipo.
- **Value constraints**: I vincoli da rispettare. Ad esempio se la proprietà deve essere compresa fra due limiti.
- **Default value**: Il valore di default. Il valore che viene automaticamente assegnato nel momento della sua creazione.
- **Auto-created**: La creazione automatica della proprietà. Nel caso in cui il nodo sia creato le proprietà con tale dicitura anche se non conterranno valori verranno create automaticamente.
- **Mandatory**: Dichiara l'obbligatorietà. La presenza di tale informazione permette al repository di richiedere la proprietà, se questa non verrà inserita, non verrà immagazzinato nemmeno l'elemento che si desiderava aggiungere.
- **OnParentVersion**: Lo stato onParentVersion della proprietà. Questo specifica che cosa succede alla proprietà, se viene creata una nuova versione del nodo padre.
- **Protected**: La proprietà protetta. Tale proprietà non potrà mai essere modificata o rimossa dall'utilizzatore del repository.
- **Multiple values**: Valori Multipli. Significa che la proprietà può avere più valori, il che vuol dire che avrà una matrice di valori e non un unico valore.
- **Child Node Definitions**: La definizione di alcuni nodi figli, che a loro volta utilizzano le proprietà in precedenza elencate.

3.6.5 Ereditarietà fra i tipi di nodi

Un tipo di nodo può avere uno, o in alcune implementazioni, più di uno, supertipo. Un sottotipo eredita le proprietà e le definizioni dal suo supertipo, e può dichiarare ulteriori proprietà.

La scoperta dei tipi di nodo disponibili nel repository avviene tramite l'oggetto NodeTypeManager, acquisito tramite il workspace. Il NodeTypeManager provvede a ricavare i nodi e a capire se questi sono primary, mixin o semplicemente elenca tutti i nodi presenti.

3.6.6 I tipi di nodi predefiniti

Ogni repository supporta almeno il tipo di nodo primario nt:base. Tutti gli altri tipi di nodo primario devono essere sottotipi di nt:base. Un certo numero di tipi predefiniti sono definiti nodi primari per i domini di applicazione comune. Sono definiti anche i tre tipi di nodo mix:referenziable, mix:versionable e mix:lockable. Ogni nodo nel momento in cui viene definito dovrà possedere la struttura generale presentata in Appendice A alla sezione A.1.

3.6.7 Definizione dei tipi di nodi nel content

E' facoltativo per il repository esporre le definizioni dei suoi tipi di nodi disponibili. Tuttavia, se espone queste definizioni, allora permette la costruzione di altri tipi di nodo *nt:nodeType* (e dei suoi tipi di nodi connessi nt:propertyDefinition e nt:childNodeDefinition). Questi tipi di nodi sono definiti per memorizzare le definizioni di altri nodi. Ad esempio, per memorizzare un nodo PropertyDefinition viene utilizzato il tipo nt:propertyDefinition. In Appendice A verranno descritti i nodi predefiniti e nei livelli successivi si capirà anche il loro utilizzo.

3.6.8 Tipi di nodi predefiniti mixin

I tre tipi mixin predefiniti sono mix:lockable, mix:referenceable e mix:versionable. Il mix:versionable è un sottotipo di mix:referenceable. mix:referenceable e mix:lockable non hanno supertipi. Non vi è alcun supertipo richiesto per i tipi di nodo mixin. In Appendice A nelle tre sezioni: A.2, A.3, A.4 vengono presentati i tipi di nodo mixin.

3.6.9 Tipi di nodi primari predefiniti

Ogni nodo del repository deve essere almeno del tipo nt:base. Di consuetudine, la creazione di nuovi nodi di tipo primario implica che il padre sia nt:base. nt:version e nt:versionHistory sono necessari per il supporto delle versioni. nt:nodeType, nt:propertyDefinition e nt:childNodeDefinition sono necessari se i nodi con il loro contenuto devono essere archiviati nel repository.

3.6.10 Gerarchia di ereditarietà dei nodi

In seguito viene creata una gerarchia di nodi che mette in evidenza l'ereditarietà fra i nodi.

nt:base
|
|–**nt:unstructured**
|
|–**nt:hierarchyNode**
| |
| |–**nt:file**

| |
| |–**nt:linkedFile**
| |
| |–**nt:folder**
|
|–**nt:nodeType**
|
|–**nt:propertyDefinition**
|
|–**nt:childNodeDefinition**
|
|–**nt:versionHistory***
|
|–**nt:versionLabels**
|
|–**nt:version***
|
|–**nt:frozenNode**
|
|–**nt:versionedChild**
|
|–**nt:query**
|
|–**nt:resource***

I nodi segnati dal simbolo * hanno, anche, come supertipo mix:referenceable.

In Appendice A dalla sezione A.5 alla sezione A.20 verranno riportate le strutture di ciascun nodo presentato nella gerarchia appena esaminata.

3.6.11 Come vengono memorizzati i nodi

La locazione */jcr:system* è riservata per immagazzinare i nodi e può essere vista come un 'sistema di cartelle'. Alcune applicazioni, ad esempio JCRBrowser (plugin per Eclipse), permettono di vedere il contenuto interno del repository strutturato in cartelle. I tipi di nodi esposti in precedenza, e i nuovi nodi che a Livello 2 si potranno definire, vengono immagazzinati in

/jcr:system/jcr:NodeTypes

Se il repository supporta le versioni, queste verrano inserite in

/jcr:system/jcr:versionStorage

3.6.12 Controllo dell'accesso

Nei casi più semplici, l'implementazione in realtà non supporta il controllo dell'accesso, quindi tutti possono accedere al repository. Tuttavia, le specifiche JSR-170 non definiscono i meccanismi per la definizione delle politiche del controllo degli accessi.

Se viene utilizzato un meccanismo esterno per l'attuazione del controllo della sicurezza un possibile candidato è rappresentato dalle JAAS Java Authentication and Authorization Service.

3.7 A Livello 2

La sezione seguente spiega il Livello 2 del Content Repository, il quale definisce un repository in lettura e scrittura. I metodi per la scrittura di contenuti nel repository possono dividersi in due categorie: quelli che scrivono direttamente nel workspace e quelli che memorizzano in modo volatile nella sessione associata. Questi ultimi richiedono un salvataggio nel momento in cui si desidera archiviare permanentemente il contenuto.

La finalità del deposito temporaneo nel corso della sessione è quello di fornire uno spazio in cui è possibile effettuare tanti cambiamenti senza convalidarli ad ogni passo, ma salvandoli in un'unica volta alla fine. I metodi che scrivono in modo volatile sono:

Node.addNode Node.setProperty Node.orderBefore Node.addMixin
Node.removeMixin Property.setValue Item.remove Session.move
Session.importXML Query.storeAsNode

Le modifiche apportate con questi metodi verranno applicate nel momento in cui verrà effettuato un salvataggio.

Session.save()

rende permanenti tutte le modifiche in sospeso attualmente memorizzate nell'oggetto Session. Al contrario,

Session.refresh(false)

elimina tutte le modifiche in sospeso attualmente memorizzate nella sessione.

I metodi invece che scrivono direttamente nel workspace sono:

Node.checkin Node.checkout Node.restore Node.restoreByLabel Node.merge
Node.cancelMerge Node.doneMerge Node.update Node.lock Node.unlock
Workspace.move Workspace.copy Workspace.clone Workspace.restore
Workspace.importXML VersionHistory.addVersionLabel
VersionHistory.removeVersionLabel VersionHistory.removeVersion.

3.7.1 Aggiunta e rimozione di nodi e proprietà

Per aggiungere un nuovo nodo al repository è necessario invocare il metodo addNode. Se viene invocato il metodo

Node.addNode(relPath)

verrà creato un nodo di tipo nt:unstructured nel path specificato, se invece il metodo chiamato è

Node.addNode(relPath, String primaryNodeTypeName)

verrà creato un nodo nel path assegnato e conforme al tipo di nodo richiesto dalla firma del metodo.

Una volta creato il nodo sarà possibile assegnargli delle proprietà con il metodo

Node.setProperty(String nameProperty, Value valueProperty)

se la proprietà è a valore singolo, se invece la proprietà è multivalore allora si avrà un array di Value come valori. Sarà anche possibile cambiare il valore di una proprietà quando è già stata immagazzinata con

Node.getProperty(nameProperty).setValue(tipo di dato)

dove nella firma del metodo può essere inserito qualsiasi tipo di dato.

La rimozione dei nodi avviene con il metodo

remove()

Prima di invocare tale metodo bisogna iterare i nodi con

getNode()

e posizionarsi nel nodo da eliminare ed in seguito chiamare il metodo per la rimozione. Se il nodo da eliminare contiene dei riferimenti o esso stesso è riferito in altri nodi, sarà obbligatorio eliminare prima tutti i riferimenti altrimenti il nodo non sarà eliminato in quanto è garantita l'integrità referenziale. Se il nodo contiene delle versioni, queste verranno mantenute nello storico.

Anche una proprietà può essere eliminata impostando con

setValue()

il valore a null oppure rimuovendo sia il nome della proprietà che il suo valore invocando il metodo

Node.getProperty(nameProperty).remove()

3.7.2 Aggiunta e rimozione di Namespace

Come già spiegato a Livello 1 il namespace permette di mappare i prefissi utilizzati dai nodi. A Livello 2 sarà possibile mappare dei nuovi namespace ed in seguito creare dei nuovi tipi di nodo personalizzati. Per registrare i namespace si utilizza

NamespaceRegistry.registerNamespace(String prefix, String uri)

Quando sarà stata fatta la registrazione del namespace sarà possibile anche registrare il nodo. E' possibile anche deregistare il namespace usando

NamespaceRegistry.unRegisterNamespace(String prefix)

non sarà possibile eliminare i namespace predefiniti nel repository, nè i namespace inseriti dal cliente e in uso nella definizione di nodi presenti nel repository.

3.7.3 Importazione XML

Il repository a Livello 2 deve sostenere l'importazione di contenuti da entrambi gli standard di mappatura del linguaggio XML: vista dal sistema e vista dal documento.

3.7.4 Assegnare tipi di nodo ad un nodo

Le implementazioni a Livello 2 devono sostenere l'assegnazione dei tipi di nodi primari e mixin nel momento della creazione del nodo e, facoltativamente, l'assegnazione e la rimozione del tipo mixin dai nodi già esistenti.

Quando un nodo viene creato, automaticamente gli viene assegnata la proprietà jcr:primaryType se è un tipo di nodo primitivo e jcr:mixinTypes se è di tipo mixin.

3.7.5 Assegnare un tipo di nodo primario

L'assegnazione di un tipo di nodo primario viene effettuata mediante la chiamata

Node.addNode(String relPath, String primaryNodeTypeName)

alternativamente in alcuni casi è possibile che non sia fornita la stringa del tipo di nodo, è sufficiente anche

Node.addNode(String relPath)

3.7.6 Assegnare un tipo di nodo mixin

Per assegnare un tipo di nodo mixin è necessario invocare

Node.addMixin(String mixinName)

il quale potenzierà il tipo di nodo primitivo con delle specifiche proprietà riguardanti le revisioni, il bloccaggio, i riferimenti in base al mixinName scelto. Anche la rimozione del tipo mixin con

Node.removeMixin(String mixinName)

sarà possibile, solamente se il tipo mixin non risulta utilizzato.

3.8 Blocchi Funzionali

Questa sezione fornisce una panoramica delle funzionalità opzionali che possono essere supportate da un'implementazione di un Content Repository. Le funzionalità aggiuntive sono: transazioni, revisioni, osservazioni, bloccaggio e ricerca mediante la sintassi SQL. Nessuna di queste caratteristiche dipende dall'altra nè da qualsiasi elemento di Livello 2, quindi una qualsiasi combinazione di queste cinque funzionalità può essere sostenuta sia da un repository di Livello 1 o di Livello 2. Come le sezioni precedenti, questa sezione è suddivisa in argomenti di base sulle categorie funzionali.

3.8.1 Transazioni

Una transazione è una sequenza di operazioni, che può concludersi con successo o meno; in caso di successo, il risultato delle operazioni deve essere permanente, mentre in caso di insuccesso si deve tornare allo stato precedente, all'inizio della transazione.

Se un Content Repository supporta le transazioni rispetta le specifiche Java Transaction. Se un'implementazione di un repository supporta le transazioni, lo si può sapere invocando il metodo

Repository.getDescriptor(OPTION_TRANSACTIONS_SUPPORTED)

il valore di ritorno true informerà che il repository è stato creato con il supporto delle transazioni.

I metodi per il controllo delle transazioni non vengono definiti da queste specifiche ma dalle JTA. Le JTA prevedono due approcci generali per le transazioni, far gestire queste dal repository o dall'utente. Nel primo caso, le transazioni vengono curate e gestite dal server dell'applicazione. Le interfacce JTA:

javax.transaction.TransactionManager
e
javax.transaction.Transaction

sono rilevanti in questo contesto.

Nel secondo caso, le transazioni vengono gestite dall'utente, l'applicazione utilizzando le specifiche JTA può scegliere di controllare i confini della transazione dall'interno dell'applicazione. In questo caso l'interfaccia di riferimento è

javax.transaction.UserTransaction

Questa è l'interfaccia che fornisce i metodi per gestire le transazioni. Da notare che dietro le quinte vengono utilizzate le interfacce

javax.transaction.TransactionManager
e
javax.transaction.Transaction

Il cliente non utilizzerà mai direttamente le due interfacce citate in entrambe le proposte di gestione delle transazioni.

3.8.2 Versioni

Un Content Repository può supportare il controllo delle versioni. Questa funzione consente ad un nodo in precedenza registrato di essere successivamente ripreso e revisionato ovviamente tenendo memorizzati tutti i precedenti salvataggi. Il sistema di versioning è stato per la prima volta definito nelle specifiche JSR-147, ora inglobate nelle JSR-170.

Repository.getDescriptor(OPTION_VERSIONING_SUPPORTED)

riferisce se il repository gestisce o meno le versioni.

Un archivio delle versioni ha una speciale area riservata all'interno del repository dove immagazzinare lo storico delle versioni. I nodi che sono stati resi versionable sono contenuti nel workspace e vengono messi in relazione mediante il proprio UUID all'area che contiene lo storico delle versioni di ciascun nodo. Lo storico è una raccolta di versioni, le quali sono collegate l'una all'altra mediante le proprietà di relazione predecessore e successore. Una nuova versione viene aggiunta in base alla cronologia delle versioni di un nodo versionable nel momento in cui viene effettuato il check-in al nodo. Con il metodo check-out un nodo viene 'aperto' e quindi modificato. Ogni nuova versione è collegata alla cronologia delle versioni mediante la proprietà successore. Il risultato è che la storia delle versioni rappresenta un grafico aciclico delle versioni in cui gli archi nel grafico rappresentano la relazione di successore. Le versioni immagazzinate sono anch'esse definite come nodi. Se vi è solo un'area riservata alle versioni nel repository, questo si riflette in ogni workspace come uno speciale, protetto, sottoalbero di nodi del tipo nt:versionHistory e nt:version.

Quando un nodo versionable è check-in

Node.checkin

una nuova versione verrà creata nello storico delle versioni per quel nodo. Il nodo versionable è impostato ad essere di sola lettura. Per 'alterare' il contenuto del metodo con un normale metodo di scrittura, deve essere effettuato il check-out con

Node.checkout

Per rendere un nodo versionable, subito dopo la creazione è necessario aggiungere con

Node.addMixin(mix:versionable)

la possibilità di creare revisioni nel futuro. Se il nodo è stato dichiarato versionable è anche automaticamente referenceable. A differenza di un nodo creato con i tipi di nodo primari, questo contiene in più le proprietà jcr:baseVersion, jcr:isCheckedOut, jcr:predecessors, jcr:mergeFailed e jcr:versionHistory.

3.8.3 Storico delle versioni

Lo storico delle versioni è costituito da un unico nodo nt:versionHistory, il quale contiene un insieme di nodi figli di tipo nt:version, che rappresentano tutte le versioni del nodo. Un nt:versionHistory ha almeno un figlio, il nodo nt:version, che rappresenta la versione principale. Da questa versione radice il grafico procede attraverso una rete di proprietà di tipo REFERENCE che collegano qualsiasi ulteriore figlio nt:version al grafico mediante le relazioni di successione.

Tutte le versioni vengono immagazzinate nella versionHistory che si trova in

/jcr:system/jcr:versionStorage

ed è di sola lettura. Per ricavare le versioni immagazzinate nello storico o si interroga la versionHistory con una query o si utilizzano i metodi presenti nell'interfaccia VersionHistory uniti a quelli di Version.

3.8.4 Le specifiche per le versioni

Le specifiche per il controllo delle versioni consistono nel connettere i nodi con le rispettive versioni mediante le due interfacce VersionHistory e Version che estendono l'interfaccia Node. VersionHistory è l'interfaccia per i nodi del tipo nt:versionHistory e Version per nt:version. Con esse sarà possibile ottenere tutte le versioni immagazzinate.

Un oggetto VersionHistory fornisce l'interfaccia per un nodo nt:versionHistory. Esso offre un comodo accesso allo storico delle versioni. Per ottenere lo storico sarà necessario invocare il metodo

getVersionHistory()

dal nodo di cui si necessita conoscere tutte le versioni ad esso associate. Alcuni utili metodi di javax.jcr.version.VersionHistory sono:

- **getVersionableUUID()**: Restituisce l'UUID del nodo versionable al quale è assegnato lo storico delle versioni.
- **getRootVersion()**: Ritorna la versione radice dello storico delle versioni.

- **getAllVersions()**: Restituisce un iteratore di tutte le versioni dello storico. Al minimo è possibile che esista un'unica versione, la versione radice. Quindi l'iteratore avrà valore uno.
- **getVersion(String versionName)**: Recupera una particolare versione dallo storico in base al nome della versione. Lancia una *VersionException* se la versione specificata non è presente.

3.8.5 Osservazione degli eventi

Un repository può supportare le osservazioni. Queste funzionalità consentono di registrare gli eventi che descrivono le modifiche ad un workspace, e quindi monitorare e rispondere agli eventi.

Repository.getDescriptor(OPTION_OBSERVATION_SUPPORTED)

permette di capire se il repository è abilitato a supportare le osservazioni.

Si noti che (nei repository che supportano le transazioni) nel caso di modifiche apportate all'interno di una transazione, il corrispondente evento non sarà spedito sul commit della transazione, mentre nel caso di modifiche effettuate al di fuori di una transazione l'evento verrà inviato nel momento in cui verrà effettuato il salvataggio. Un oggetto che implementa l'interfaccia Event rappresenta un evento generato dal repository.

3.8.6 Lock

Un repository che implementa il bloccaggio, Lock, permette a un utente di bloccare temporaneamente i nodi al fine di impedire ad altri utenti di cambiarli. Questa funzione viene in genere utilizzata per serializzare l'accesso ad un nodo e per impedire la sovrascrittura involontaria del contenuto presente nel nodo. Per capire se è implementato si invoca il metodo

Repository.getDescriptor(OPTION_LOCKING_SUPPORTED)

Un blocco è posto su un nodo chiamando

Node.lock

Solo i nodi mix:lockable (ereditato o esplicitamente assegnato) possono bloccare il contenuto. Un blocco può essere superficiale se riguarda soltanto il singolo nodo, o profondo se contiene oltre al nodo anche i nodi del sottoalbero ovvero i figli. L'utente che inserisce un blocco su un nodo è il proprietario del blocco. Il proprietario del blocco è identificato da l'ID utente associato alla sessione attraverso il quale il nodo ha subito il lock (cioè, la stringa restituita da Session.getUserID). L'ID utente è registrato nella proprietà jcr:LockOwner del nodo lock. E' possibile rimuovere il blocco dal nodo in qualsiasi istante invocando il metodo

Node.unlock

da parte dell'utente che aveva generato il blocco.

3.8.7 Query sintassi SQL

Un repository può supportare anche la sintassi SQL, la ricerca viene effettuata usando gli stessi metodi già proposti a Livello 1. Ovviamente non è molto consigliato l'uso del linguaggio SQL in quanto essendo il repository una struttura ad albero la sintassi XPath lavora meglio e con una velocità maggiore [15].

Capitolo 4

Apache Jackrabbit

In questo capitolo si tratterà Apache Jackrabbit, implementazione di riferimento delle specifiche JSR-170.

Le ragioni principali che hanno portato Sanmarco Informatica a scegliere Jackrabbit sono principalmente quattro:

- Open Source
- Multipiattaforma
- Stabilità
- Implementazione di riferimento per le specifiche JSR-170 Java Content Repository

Apache Jackrabbit è l'unico progetto su cui poter implementare un proprio Content Repository basato sulle esigenze aziendali.

Verrà presentato in seguito il modello Standalone Server, primo approccio utilizzato per capire la struttura e il funzionamento di Jackrabbit. Durante lo sviluppo del progetto, è stato abbandonato il modello Standalone Server, in quanto l'attenzione si è spostata sull'accesso al Repository mediante metodi di invocazione remota (RMI), e tale Server risultava limitato quindi è stato sostituito con il Server Apache Tomcat. Saranno altresì analizzati, sempre in questo capitolo, i componenti, l'architettura Jackrabbit e la configurazione.

4.1 Standalone Server

Apache Jackrabbit [16] è un progetto Open Source ed è un'implementazione completa delle API JCR, tra l'altro in esso sono presenti le specifiche primarie per sviluppare l'applicazione. Apache Jackrabbit è pienamente compatibile con le JSR-170. Al di là delle specifiche JSR-170, Jackrabbit contiene numerose estensioni e funzioni amministrative necessarie per gestire repository.

Ogni versione Jackrabbit viene rilasciata con un'applicazione di base precostruita e con un file jar eseguibile, che lanciato fa partire il Server Standalone;

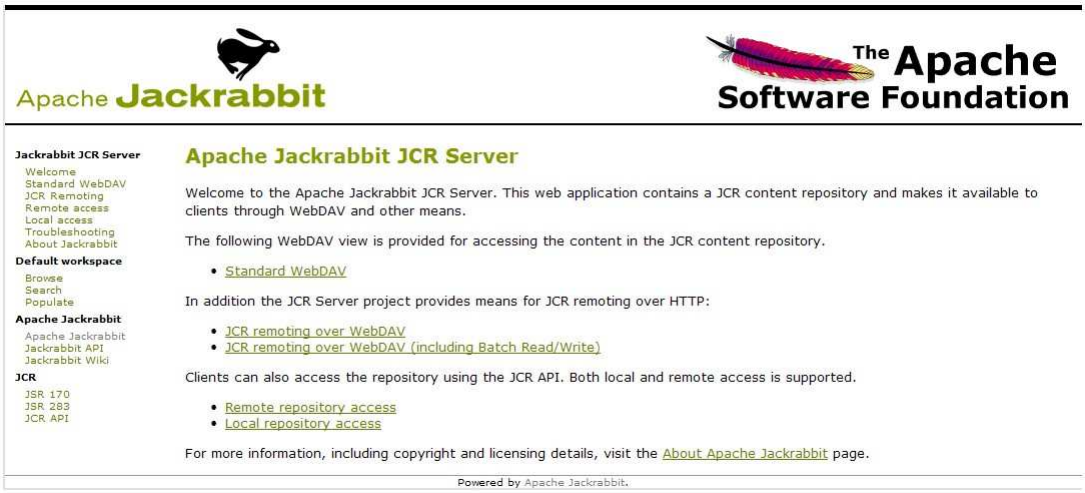

Figura 4.1: Server attivo.

il Repository Jackrabbit è attivo. E' la via più semplice per iniziare a capire Jackrabbit. E' necessaria una versione Java 5 o superiore. Il server si lancia da linea di comando come riportato in seguito:

java -jar jackrabbit-standalone-1.6.0.jar

Dopo aver lanciato il server, è possibile vedere Jackrabbit in azione su

http://localhost:8080/

Per cambiare la porta di default 8080, con un'altra porta, basta usare l'opzione - *-port* da riga di comando e specificare la diversa porta desiderata. Dal browser si dovrebbe vedere il server della Figura 4.1.

Di default, il Server creerà una directory *./jackrabbit* e un file di configurazione *repository.xml* in essa contenuto. La cartella jackrabbit e la configurazione repository.xml sono automaticamente create se non esistono, se già sono presenti vengono caricate quelle già esistenti. I *file di log* vengono scritti nella sottodirectory di log del repository. E' possibile spegnere il server premendo *Ctrl-C*. Il server verrà chiuso e il contenuto del repository rimarrà immutato se in precedenza non è stato salvato. E' molto importante salvare, perché se in una sessione non si salvano gli aggiornamenti il repository conterrà le informazioni presenti al momento dell'attivazione del server.

E' consigliabile aumentare la quantità di memoria della macchina virtuale Java se si prevedono accessi concorrenti al repository o se si devono effettuare operazioni batch di grandi dimensioni.

L'interfaccia web che si vede su

http://localhost:8080/

è una semplice applicazione costruita sul Content Repository. Essa contiene alcune istruzioni di base su come accedere al repository e vengono fornite alcune utilità generali come:

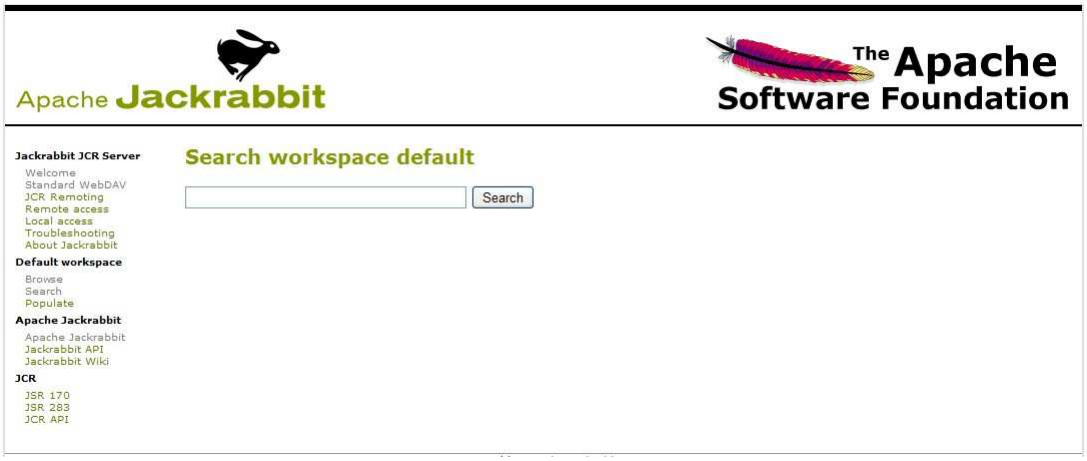

Figura 4.2: Ricerca.

- **Inserimento:** inserisce dei documenti trovati dal web.
- **Ricerca:** utilizza la funzionalità di ricerca full text su tutti i documenti contenuti nel repository. Come si può vedere nella Figura 4.2
- **Visualizzazione:** visualizza tutti i contenuti del repository sottoforma di cartelle (nt:folder), sottocartelle e file (nt:file), detti nodi. Per accedere al Repository contenuto nel Server si può usare qualsiasi utente, non è abilitato il controllo dell'accesso. La Figura 4.3 illustra quanto spiegato in questo punto.

Il contenuto del repository si può vedere semplicemente dal browser all'indirizzo

http://localhost:8080/repository/default/

questo è in realtà un server WebDAV (Figura 4.4) completamente descritto e sostenuto dal Content Repository.

Ovviamente tale accesso è limitato alla visualizzazione dei singoli documenti. Per una visualizzazione più accurata, intesa come visualizzazione delle proprietà/attributi, si può accedere all'indirizzo

http://localhost:8080/server

E' necessario implementare una serie di funzionalità avanzate in WebDAV o nel Client HTTP, e con il componente Jackrabbit spi2dav si potrà avere il pieno accesso al JCR remoto.

Nello sviluppo del progetto richiesto dall'azienda verrà posto un limite all'utilizzo di WebDAV. Quest'ultimo verrà utilizzato soltanto dall'indirizzo

http://localhost:8080/reposity/default/

dove qualsiasi client può visualizzare il contenuto. Per analizzare tutte le proprietà di un nodo verrà installato un plugin in Eclipse chiamato JCRBrowser che sostituisce la visualizzazione accurata che offre WebDAV all'indirizzo

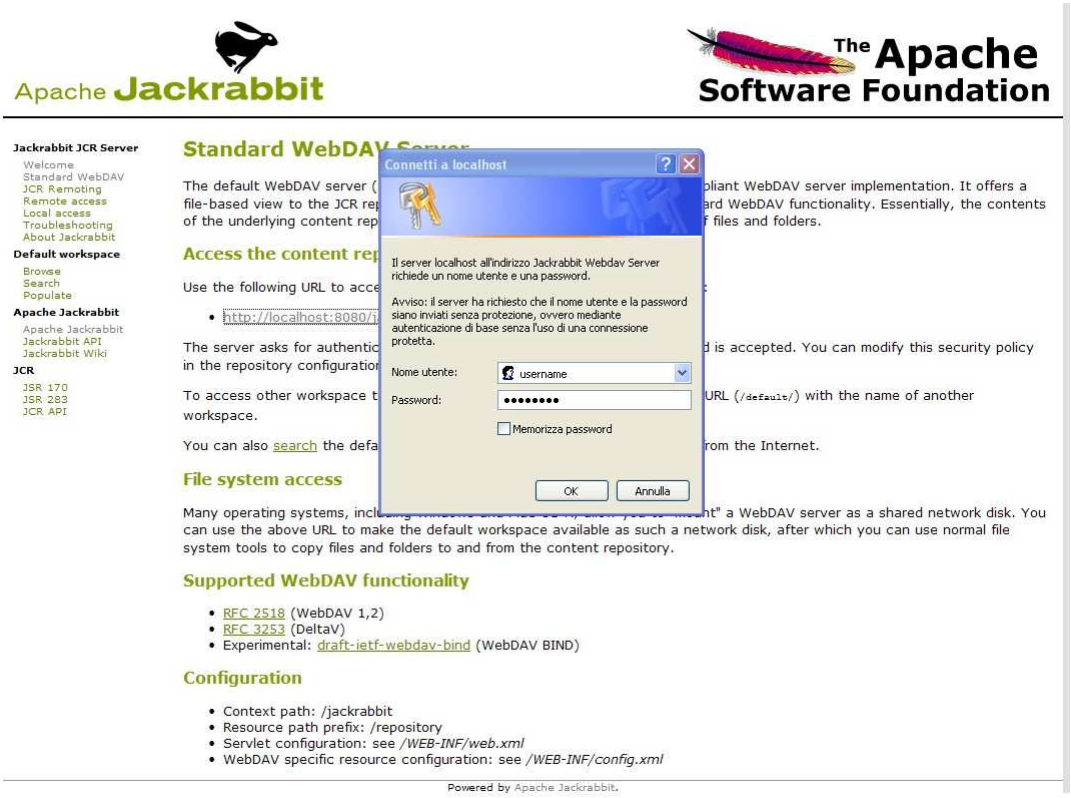

Figura 4.3: Accesso al Repository.

http://localhost:8080/server

In aggiunta al WebDAV Server, il Server Standalone supporta anche l'accesso remoto con le specifiche JCR-RMI. Dopo aver aggiunto i sorgenti JCR-RMI è possibile accedere al repository remoto nel seguente modo:

Repository repository = new URLRemoteRepository(http://localhost:8080/rmi);

E' inoltre possibile eseguire il backup dell'applicazione, quando il repository non è in esecuzione, con il comando sotto riportato:

java -jar jackrabbit-standalone-1.6.0.jar - - backup

Se ci fosse bisogno di migrare ad una configurazione diversa di repository, grazie al comando - *-backup-conf* è possibile effettuare il cambiamento.

Il Server Standalone è stato progettato principalmente come un modo rapido e semplice per ottenere un Content Repository, ma soprattutto per studiare e per eseguire test di sviluppo. Da notare inoltre che il livello JCR-RMI non è ottimizzato per tutte le prestazioni. L'azienda utilizza tale livello, anche se non ottimizzato alla perfezione, perché si è visto che tutte le funzionalità richieste per la gestione documentale vengono supportate.

Questa sarà la soluzione adottata: installare Tomcat in un elaboratore che svolge la funzione di server e fare il deploy di *jackrabbit.war* pacchetto che contiene

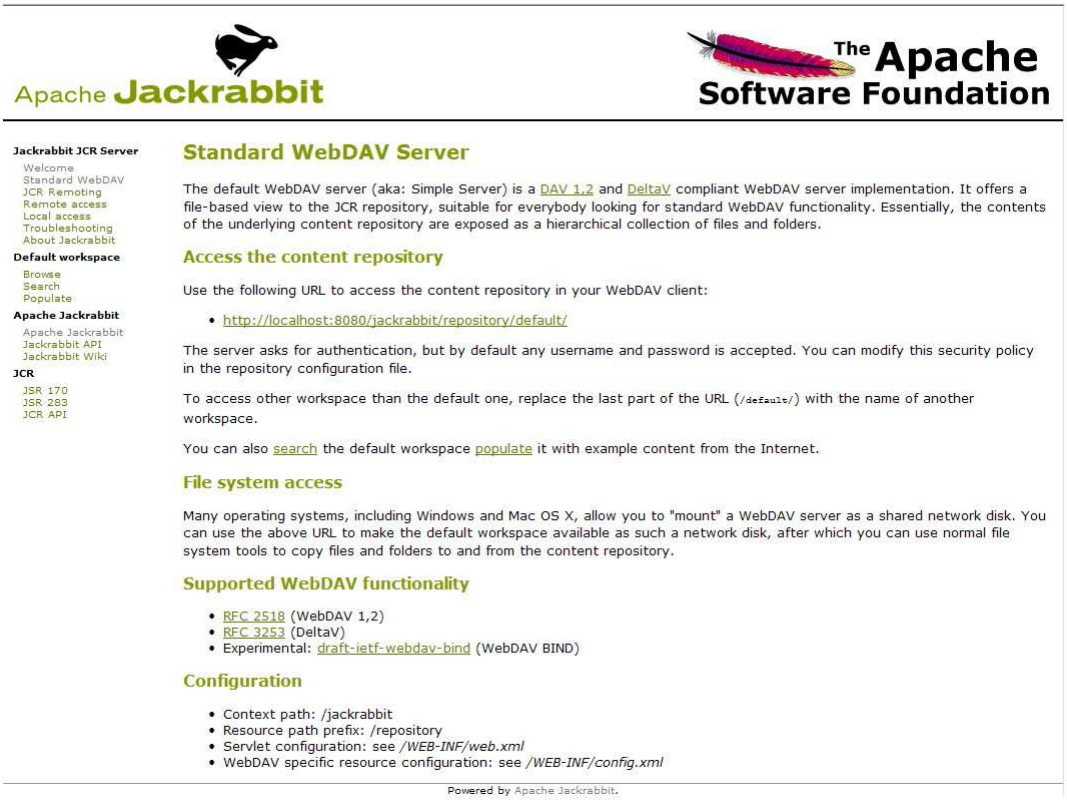

Figura 4.4: Standard WebDAV Server

le specifiche standalone e rmi. A questo punto nell'elaboratore che funge da server attivare il server Tomcat il quale manderà in esecuzione il repository. Da un qualsiasi client remoto, mediante il supporto JCR-RMI, è possibile loggarsi e aprire una sessione con il repository, oppure semplicemente vedere il contenuto dal browser all'indirizzo

http://ipserver:porta/jackrabbit

4.2 Componenti Jackrabbit

Il progetto Apache Jackrabbit è costituito da un numero di componenti [17] connessi tra loro, che sono:

- **Jackrabbit API:** contiene le interfacce che si possono estendere per aggiungere funzionalità agli standard JCR-API. E' possibile usare queste interfacce per accedere a specifiche funzionalità.
- **Jackrabbit JCR Commons:** contiene un certo numero di classi con finalità generali per l'utilizzo di JCR-API.
- **Jackrabbit JCR Test:** contiene dei test per accertare la conformità della realizzazione.

- **Jackrabbit Core:** contiene il nucleo delle specifiche JSR-170.
- **Jackrabbit TextExtractor:** contiene le classi che permettono di estrarre il contenuto del testo da file binari e poter così attuare l'indicizzazione full text.
- **Jackrabbit JCR-RMI:** contiene le classi per accedere da remoto al repository JCR.
- **Jackrabbit WebDAV Library:** fornisce interfacce e classi utilizzate per la costruzione di un sever o di un client WebDAV.
- **Jackrabbit JCR Server:** contiene due implementazioni basate su server WebDAV.
- **Jackrabbit JCR Servlet:** contiene una serie di servlet per rendere più semplice l'utilizzo del repository nell'interfaccia web.
- **Jackrabbit WEB Application:** fornisce servlet per accedere a un repository Jackrabbit.
- **Jackrabbit JCA Resource Adapter:** permette al Content Repository di adattarsi in una qualsiasi applicazione di rete.
- **Jackrabbit SPI:** permette una separazione tra i componenti transitori e persistenti del Repository. La componente transitoria (client repository) viene attuata sopra alla componente persistente (server repository).
- **Jackrabbit SPI Commons:** contiene classi per l'implementazione di Jackrabbit SPI.
- **Jackrabbit JCR to SPI e SPI to JCR:** contengono entrambi l'implementazione delle SPI interfacce.
- **Jackrabbit Standalone Server:** già ampiamente spiegato in precedenza.
- **Jackrabbit OCM e OCM Node Management:** le classi qui contenute permettono di mappare i contenuti del Repository.

Tutti questi componenti sono considerati stabili. Nel progetto che verrà sviluppato si utilizzerà il componente Jackrabbit JCR-RMI. Il componente Jackrabbit Text-Extractor è utilizzato di default.

Tutte le operazioni più richieste sono definite nelle specifiche JCR, come già ampiamente spiegato nel Capitolo 3. Alcune funzionalità, ad esempio di amministrazione e accesso al repository, non sono ancora ben definite nelle JSR-170. I componenti di Jackrabbit integreranno tali mancanze.

Apache Jackrabbit contiene tutte le funzionalità descritte e si presta bene come infrastruttura su cui costruire un proprio Content Repository o per una qualsiasi applicazione che necessita di memorizzare determinate informazioni.

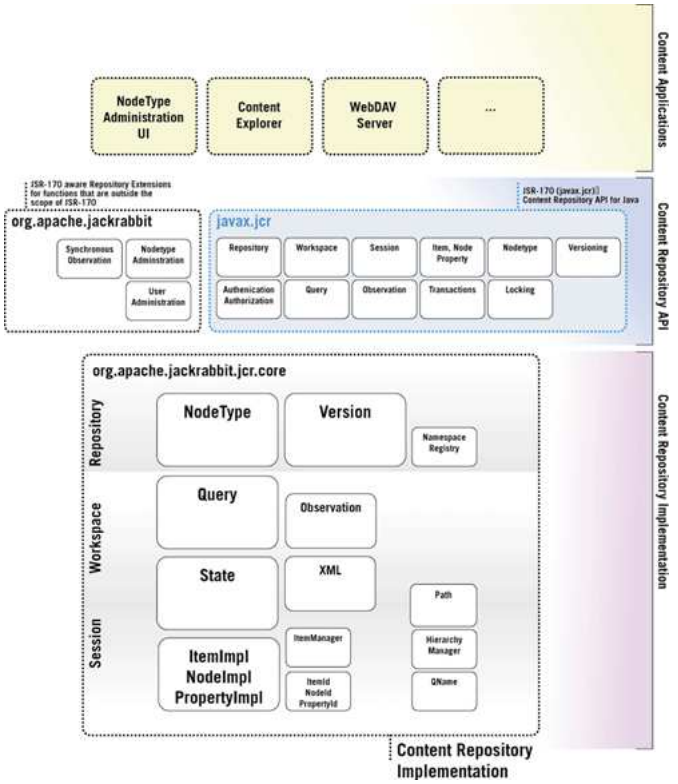

Figura 4.5: Livelli Architettura Jackrabbit.

4.3 Architettura Apache Jackrabbit

In questo paragrafo, riguardante l'architettuta Jackrabbit [18], verrà definito fisicamente Apache Jackrabbit.

L'architettura generale di Jackrabbit, come si può vedere dalla Figura 4.5, è definita in tre livelli:

- **Content Application Layer.** Interagisce attraverso le specifiche JSR-170 con il Content Repository. Ci sono delle applicazioni disponibili per le specifiche JSR-170, alcune molto generiche come un WebDAV Server, altre invece molto specifiche le quali fanno uso di un Content Repository per archiviare informazioni. Le applicazioni Java possono utilizzare le JSR-170 per creare Content Repository per immagazzinare qualsiasi tipo di informazione. Utilizzando un Content Repository è possibile beneficiare del controllo delle versioni, delle query, delle relazioni e molto altro ancora che lo rende un archivio dati ideale per molte applicazioni. Il Content Application Layer è il livello più alto, lo si può definire come un'interfaccia utilizzata per osservare le informazioni.

- **Content Repository API.** E' suddiviso in due sezioni principali:

  – Content Repository API definite dalle specifiche JSR-170.

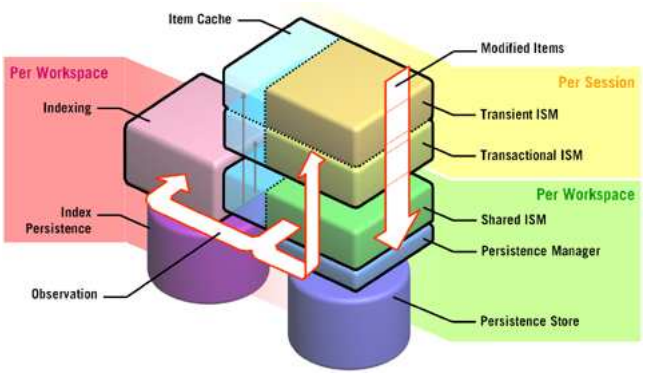

Figura 4.6: Funzionamento Jackrabbit.

  - Altre caratteristiche di Content Repository non definite nelle specifiche JSR-170 ma in altri pacchetti, per la maggior parte suddivisi in blocchi di funzionalità.

- **Content Repository Implements Layer.** Riflette gli elementi principali della realizzazione di un Content Repository Jackrabbit. Nella Figura 4.5 la dimensione dei blocchi simboleggia la quantità di codice e quindi la complessità del singolo blocco funzionale. Ci sono tre ambiti in un Content Repository: il Repository stesso, il Workspace e la Session. Ogni funzione può essere attribuita ad uno o più blocchi. Questo livello contiene le componenti più importanti per la realizzazione del Content Repository.

4.4 Come lavora Jackrabbit

La Figura 4.6 illustra quali componenti di Jackrabbit sono utilizzati quando un utente modifica il contenuto del Content Repository. Questa è un'operazione semplice e molto comune, che tocca gran parte dei componenti utilizzati per l'implementazione di Jackrabbit. Da tener presente che questa architettura è stata progettata sulla base delle specifiche JCR. In seguito saranno illustrati i componenti e le rispettive funzioni, nel caso di modifiche al contenuto del Repository [19].

- **Transient Item State Manager.** Quando gli elementi vengono letti da una sessione vengono tenuti nella Transient ISM e quando vengono modificati, la modifica è visibile solo in quella stessa sessione.

- **Transactional Item State Manager.** Quando si desidera immagazzinare le nuove informazioni deve essere invocato un salvataggio nella sessione con Session.save(), così i dati transitori vengono immagazzinati nella Transactional ISM. Le modifiche sono ancora visibili solo nella presente sessione, il che significa che le altre sessioni non vedranno le modifiche fino a quando non aggiorneranno la loro sessione o non attueranno un nuovo login.

- **Shared Item State Manager.** Nel momento in cui un elemento viene condiviso, le modifiche vengono pubblicate a tutte le sessioni presenti nella stessa area di lavoro, workspace.

- **Persistence Manager.** Il Persistence Manager immagazzina tutto ciò che è stato condiviso, cioè presente nello Shared ISM. Questo gestore è un'interfaccia molto semplice e di basso livello e non ha bisogno di capire come le informazioni vengono depositate nel repository. Deve essere solamente in grado di immagazzinare e recuperare un oggetto.

- **Observation.** Quando un dato è stato reso condiviso risulta anche osservato. Si possono così avere delle osservazioni sincrone utili per le ricerche.

- **Query Manager/Index.** Attraverso un evento sincrono il Query Manager viene incaricato ad indicizzare elementi nuovi o modificati. L'indicizzazione è un'operazione interna molto complessa. E' una caratteristica propria del Content Repository, ne è esempio la ricerca full text.

Tutto questo risulta essere un meccanismo interno al JCR nascosto anche al programmatore. Il ciclo di vita di un Content Repository inizia nel momento in cui si effettua una chiamata ad un Repository, si attua il login ottenendo così una Session e da questo momento è possibile lavorare nel repository. Ogni informazione che verrà inserita seguirà il processo delle frecce bianche presenti nella Figura 4.6.

4.5 Implementazione della ricerca

Jackrabbit indicizza tutto il contenuto di un documento comprese le sue proprietà. Nella ricerca questo è altamente utile perchè sarà possibile effettuare ricerche full text grazie all'API Lucene presente nel progetto Jackrabbit. Le funzionalità di indicizzazione possono essere attuate in un qualsiasi formato di documento (PDF, HTML, documento Office, etc.). Il risultato del processo di indicizzazione è un indice interrogabile in fase di ricerca mediante query.

Jackrabbit implementa, per la creazione di interrogazioni, sia il linguaggio XPath che SQL. Pertanto la ricerca sarà indipendente dalla sintassi delle query usata, è decisamente preferibile usare XPath a causa della struttura gerarchica del JCR.

Sanmarco Informatica punterà all'utilizzo di query scritte in XPath.

4.6 Controllo della concorrenza

Il modello interno di concorrenza [20] in Apache Jackrabbit è abbastanza complesso e una serie di questioni di deadlock sono state segnalate nel momento del rilascio di Jackrabbit 1.x. Ciò che verrà scritto in questo paragrafo è il risultato di una progettazione e di una revisione mirate a prevenire problematiche

simili. Verrà analizzata la concorrenza interna e il modello di sincronizzazione in Jackrabbit. Da notare che le revisioni che hanno portato a descrivere quanto segue sono state fatte per prevenire direttamente situazioni di stallo e definire un controllo della concorrenza a livello architettonico e non su singoli componenti.

Quanto scritto in seguito si basa sulla versione di default Jackrabbit 1.5.

4.6.1 L'Architettura di fondo

L'architettura Jackrabbit può essere divisa in cinque livelli:

- **Cluster.** Tale strato si occupa dei cambiamenti di sincronizzazione tra uno o più nodi del cluster che sono trattati come repository per la condivisione del contenuto. Il controllo della concorrenza attraverso i diversi nodi del cluster è realizzato utilizzando un unico blocco di scrittura. Tutti i nodi del cluster sono in grado di leggere parallelamente il contenuto condiviso con un'esplicita sincronizzazione. Da notare che i nodi condivisi in un singolo blocco portano a situazioni di stallo. Un unico nodo ancora in fase di stallo può bloccare l'intero cluster. Sviluppi futuri andranno a modificare scenari di questo genere.

- **Repository.** Il livello repository organizza la gestione del Registro di Sistema. La sincronizzazione non è globale, ma tutti i componenti hanno i loro meccanismi di sincronizzazione personalizzata. La componente più rilevante dal punto di vista del controllo della concorrenza è l'archiviazione di una versione, che contiene in realtà due meccanismi di blocco: VersionManagerImpl di alto livello per il controllo delle versioni, e SharedItemStateManager di livello più basso per il controllo dell'accesso.

- **Workspace.** Un repository è costituito da una o più aree di lavoro, workspace, le quali contengono tutti i nodi strutturati in maniera gerarchica come una struttura ad albero. Ogni workspace ha due componenti, il meccanismo di persistenza e gli indici di ricerca. Il primo è costituito da uno SharedItemStateManager che controlla le operazioni e un Persistence Manager che immagazzina gli oggetti in modo permanente. La maggior parte dei Persistence Manager utilizza la sincronizzazione Java o qualche altro meccanismo di blocco per il controllo della concorrenza, ma dato che solitamente non interagiscono molto con le altre parti del repository non sono situazioni critiche. Il secondo lo SharedItemStateManager, che utilizza un blocco di lettura-scrittura è un elemento fondamentale per il modo in cui interagisce con l'archivio dati. Jackrabbit 1.4 è configurato affinché lo SharedItemStateManager consente l'accesso in scrittura simultanea a diverse parti.

- **Session.** E' possibile accedere ad un workspace con una o più sessioni. Ogni sessione contiene uno spazio temporaneo che tiene traccia di tutte le modifiche non salvate in quella sessione. Dato che lo spazio riservato ad una sessione è locale non ci sono problemi di concorrenza.

- **Transaction.** Le transazioni sono gestite in Jackrabbit includendo tutte le operazioni (salvataggio delle modifiche, gli aggiornamenti, il controllo delle versioni) in uno spazio transitorio più ampio che persiste solo quando la transazione è impegnata.

4.6.2 Principali meccanismi di sincronizzazione

I principali meccanismi di sincronizzazione in Jackrabbit sono i blocchi di lettura-scrittura nelle classi SharedItemStateManager e VersionManagerImpl. Anche altri componenti hanno alcune funzioni di controllo di concorrenza per esempio la classe LockManagerImpl e la classe NodeTypeRegistry.

I tre blocchi principali sono:

- **Workspace lock.** Blocco di lettura-scrittura per SharedItemStateManager (nel workspace).
- **Versioning lock.** Blocco di lettura-scrittura per VersionManagerImpl (nel repository).
- **Version store lock.** Blocco di lettura-scrittura per SharedItemStateManager associato al gestore delle versioni. Ciascun blocco può essere bloccato esclusivamente per l'accesso in scrittura o globalmente per l'accesso in lettura. In altre parole qualsiasi numero di lettori concorrenti può mantenere il blocco, ma un singolo scrittore bloccherà tutti gli altri lettori e scrittori.

4.6.3 Condizioni di deadlock

Un deadlock può verificarsi solo se il titolare di un blocco tenta di acquisire un altro blocco e non vi è un altro thread che cerca di fare il contrario. Questa situazione può verificarsi solo se: i blocchi vengono acquisiti in una sequenza nidificata, thread differenti acquisiscono blocchi nidificati in ordine diverso, almeno due blocchi esclusivi vengono acquisiti. Jackrabbit nella maggior parte delle operazioni utilizza uno dei seguenti modi per evitare deadlock: in un preciso momento viene tenuto un unico blocco, quindi non è possibile acquisire più di un blocco in una sequenza nidificata e nemmeno due blocchi esclusivi. In caso di blocchi nidificati, se il primo risulta bloccato non si passerà mai al prossimo, non sarà possibile che thread differenti acquisiscano blocchi in ordine diverso.

4.7 I tre modelli di sviluppo

Jackrabbit è costruito per supportare diverse modalità di distribuzione [21], alcune possibilità saranno di seguito proposte.

- **Web Application Bundle (Figura 4.7).** Alcune applicazioni vengono eseguite in un contesto chiuso senza interagire con altre fonti di dati, in questo caso si può utilizzare un Repository Bundle. Grazie alle specifiche

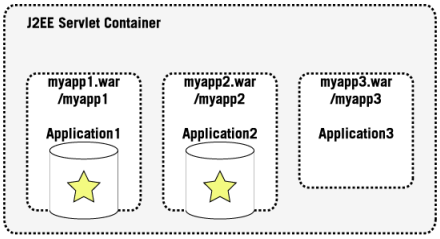

Figura 4.7: Web Application Bundle.

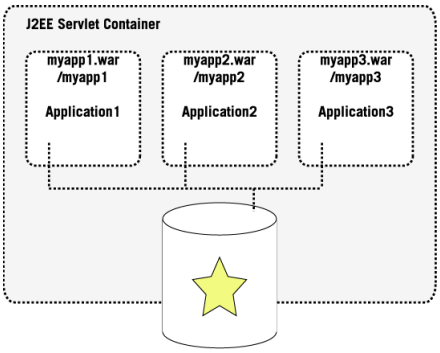

Figura 4.8: Shared J2EE Resource.

JSR-170, questo repository nel tempo può cambiare modalità di distribuzione. Il pacchetto creato è in esecuzione nella propria JVM e non sarà accessibile da altre applicazioni. Ovviamente non sarà necessario il collegamento di rete in quanto il repository sarà attivo localmente. Si ha quindi un repository locale che soltanto il client che lo possiede in memoria lo può usare. In questo modello di distribuzione si assume un'applicazione web confezionata in un file, quindi per visualizzare il contenuto basterà un browser anche senza il collegamento di rete.

- **Shared J2EE Resource (Figura 4.8).** E' un'estensione del modello precedente, il repository sarà disponibile nell'intera rete locale, intranet.

- **Repository Server (Figura 4.9).** In ambienti aziendali il modello di distribuzione client/server è ampiamente utilizzato. Questo sarà il modello che verrà utilizzato anche da Sanmarco Informatica. Il progetto verrà implementato in questo modello di distribuzione. Il Content Repository può essere utilizzato da molti client nello stesso istante. Ovviamente deve essere presente un collegamento di rete Internet. In questo caso ciascun client può contribuire all'inserimento delle informazioni.

4.8 Come configurare Jackrabbit

Apache Jackrabbit [22] necessita di due parti di informazione per impostare un'istanza duratura di Content Repository:

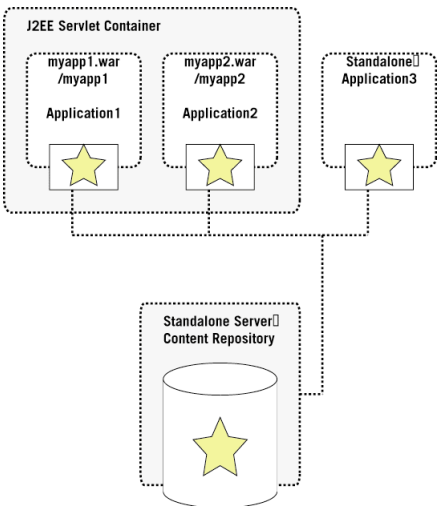

Figura 4.9: Repository Server.

- **Repository home directory.** Il path del file system della directory contenente l'accesso al Content Repository dall'istanza duratura di Jackrabbit. Questa directory usualmente contiene tutti i Content Repository, gli indici di ricerca, la configurazione interna, e altre informazioni persistenti gestite all'interno del Content Repository. Si noti che questo non è assolutamente necessario, alcuni gestori di persistenza e altri componenti Jackrabbit possono essere configurati per accedere a file o ad altre risorse al di fuori della cartella repository home directory. La repository directory home è sempre necessaria anche se alcuni componenti scelgono di non usarla. Jackrabbit compila direttamente la home directory con tutti i file necessari e le sottodirectory.

- **Repository configuration file.** Il path del file system dove si trova il file xml di configurazione del repository. Questo file specifica i nomi delle classi e le proprietà dei vari componenti Jackrabbit utilizzati per gestire e accedere al Content Repository. Jackrabbit analizza questo file di configurazione e istanzia quanto viene in esso specificato, il Content Repository viene creato.

Questi due parametri di configurazione sono passati direttamente a Jackrabbit quando si crea un'istanza di repository.

4.8.1 Configurare il Repository

Il file di configurazione del repository è repository.xml, specifica le opzioni a livello generale come la sicurezza, le versioni, la ricerca e molto altro. La configurazione di default del workspace è inclusa nella configurazione del repository. Tutte le versioni di Jackrabbit sono retrocompatibili, ciò significa che se si desidera usare una nuova versione non serve configurare nuovamente il repository. Ovviamente sarà necessario apportare delle nuove modifiche di configurazione se si desidera attivare nuove funzionalità.

```
<!DOCTYPE Repository PUBLIC
"-//The Apache Software Foundation//DTD Jackrabbit 1.6//EN"
"http://jackrabbit.apache.org/dtd/repository-1.6.dtd">
<Repository>
  <FileSystem ... />
  <Security ... />
  <Workspaces ... />
  <Workspace ... />
  <Versioning ... />
  <SearchIndex ... />
  <Cluster ... />
  <DataStore ... />
</Repository>
```

Codice 4.1: Struttura file di configurazione repository.xml

Gli elementi per configurare il repository sono:

- **File System:** Il File System virtuale utilizzato dal repository per memorizzare i dati come registrazioni del namespace e i tipi di nodo.
- **Security:** Configurazione di autenticazione e autorizzazione.
- **Workspaces:** Configurazione di dove e come i workspace sono gestiti.
- **Workspace:** Configurazione del workspace di default.
- **Versioning:** Configurazione delle versioni nel repository.
- **SearchIndex:** Configurazione degli indici di ricerca che copre tutto il repository dalla radice dell'albero */jcr:system*.
- **Cluster** Configurazione dei Cluster.
- **DataStore** Configurazione della memorizzazione dell'intero documento in formato binario. In assenza di questa clausola il documento viene segmentato all'interno del repository.

E' preferibile inserire il file di configurazione repository.xml all'interno della cartella repository home directory. Ciò mantiene il repository e la sua configurazione ben contenuta all'interno della struttura delle directory.

4.8.2 Configurare la Sicurezza

La configurazione della sicurezza è utilizzata per specificare impostazioni di autenticazione e autorizzazione per l'accesso al repository. La struttura è illustrata nel Codice 4.2.

Di default Jackrabbit utilizza le JAAS (Java Authentication and Authorization Service) per autenticare gli utenti che cercano di accedere al repository.

```
<Security appname="Jackrabbit">
  <SecurityManager ... />
  <AccessManager ... />
  <LoginModule ... />
</Security>
```

Codice 4.2: Configurazione Security.

Il paramentro appname, all'interno dell'elemento security, viene utilizzato con il nome dell'applicazione da sottoporre alle JAAS. In questo caso Jackrabbit.

Se l'autenticazione JAAS non è disponibile o troppo complessa da attuare, Jackrabbit permette di specificare un LoginModule utilizzato poi per l'autenticazione degli utenti. La classe SimpleLoginModule presente in Jackrabbit implementa un meccanismo semplice e banale di autenticazione che accetta qualsiasi nome utente e password come credenziali di autenticazione valide.

Quando l'utente è stato autenticato, Jackrabbit utilizzerà l'Access Manager configurato per controllare ciò che l'utente è autorizzato a fare. La classe SimpleAccessManager inclusa nel progetto implementa un semplice meccanismo di autorizzazione che concede accesso in lettura a tutti gli utenti e in scrittura a tutti, tranne agli utenti anonimi.

4.8.3 Configurare il Workspace

Un repository può contenere uno o più workspace che vengono configurati in un file di configurazione a parte, workspace.xml. L'elemento workspace all'interno del file repository.xml specifica dove e come è gestito il workspace. Il workspace è configurato come nel Codice 4.3.

```
<Workspaces rootPath="rep.home/workspaces" defaultWorkspace="default"/>
<Workspace ... />
```

Codice 4.3: Configurazione Workspaces.

RootPath è la directory del file system. Ogni cartella viene creata automaticamente per ciascun workspace e il percorso delle sottodirectory può variare.

DafaultWorkspace è il nome del workspace predefinito.

La configurazione del file workspace.xml che andrà richiamata all'interno del tag Workspace, sarà composta come nel Codice 4.4.

Per modificare un workspace esistente sarà necessario modificare il file workspace.xml, se viene modificato il workspace all'interno del file repository.xml la modifica sarà ininfluente.

4.8.4 Configurare le Versioni

Le versioni di un nodo versionable vengono memorizzate nel repository se è stato configurato l'elemento Versioning. La configurazione è molto simile alla confi-

```
<Workspace Name="wsp.name">
  <FileSystem ... />
  <PersistenceManager ... />
  <SearchIndex ... />
</Workspace>
```

Codice 4.4: Configurazione workspace.xml.

gurazione di un workspace, in quanto sono entrambi utilizzati per memorizzare delle informazioni. La principale differenza tra la configurazione del controllo delle versioni e quella del workspace è la SearchIndex. Una versione infatti viene ricercata utilizzando la version history. La struttura della configurazione delle versioni è illustrata nel Codice 4.5.

```
<Versioning rootPath="rep.home/version">
  <FileSystem ... />
  <PersistenceManager ... />
<Versioning/>
```

Codice 4.5: Configurazione Versioning.

4.8.5 Configurare la Ricerca

La configurazione della ricerca non avviene in modo automatico con la creazione del repository in quanto è opzionale. Si dovrà quindi modificare e aggiungere al tag SearchIndex la parte di codice scritta nel Codice 4.6.

```
<SearchIndex class="org.apache.jackrabbit.core.query.lucene.SearchIndex">
  <param name="path" value="wsp.home/index" />
  <param name="textFilterClasses"
      value="org.apache.jackrabbit.extractor.PlainTextExtractor,
          org.apache.jackrabbit.extractor.MsWordTextExtractor,
          org.apache.jackrabbit.extractor.MsExcelTextExtractor,
          org.apache.jackrabbit.extractor.MsPowerPointTextExtractor,
          org.apache.jackrabbit.extractor.PdfTextExtractor,
          org.apache.jackrabbit.extractor.OpenOfficeTextExtractor,
          org.apache.jackrabbit.extractor.RTFTextExtractor,
          org.apache.jackrabbit.extractor.HTMLTextExtractor,
          org.apache.jackrabbit.extractor.XMLTextExtractor" />
  <param name="extractorPoolSize" value="2" />
  <param name="supportHighlighting" value="true" />
  </SearchIndex>
```

Codice 4.6: Configurazione SearchIndex.

4.8.6 Configurare il Persistence Manager

Il gestore della persistenza è in realtà una delle parti più importanti della configurazione, è necessaria una notevole cura nella memorizzazione dei nodi e delle proprietà. Esistono diversi modi di memorizzazione dati, o in un database, o in un file system, quest'ultima modalità e stata scelta da Sanmarco Informatica. Nel paragrafo successivo verrà presentata la configurazione del file system.

4.8.7 Configurare il File System

L'elemento file system configura un file system virtuale. E' consigliabile utilizzare l'implementazione LocalFileSystem che associa gli accessi astratti al sistema alla directory specificata all'interno del file system originario. (Codice 4.7)

```
1 <FileSystem class="org.apache.jackrabbit.core.fs.local.LocalFileSystem">
    <param name="path" value=path/>
3 </FileSystem>
```

Codice 4.7: Configurazione File System.

4.8.8 Configurare il Datastore

Il Datastore è un elemento facoltativo, Sanmarco Informatica infatti decide di non usarlo. Permette soltanto di immagazzinare grandi valori binari, ovvero archiviare integralmente, (senza suddividerlo come fa il meccanismo interno di Jackrabbit), tutto il contenuto del documento, per migliorare le prestazioni e ridurre gli accessi al disco.

4.9 Tipi di nodo

Jackrabbit contiene un NodeTypeRegistry [23] che viene creato in fase di start-up e popolato con il set dei tipi di nodi disponibili, che includono sia quelli contenuti nelle specifiche JSR-170 e altri che possono essere inseriti dall'utente.

I nodi personalizzati si definiscono in un file di testo utilizzando la sintassi 'Compact Namespace and Node Type Definition' (CND) e registrati in seguito nel repository utilizzando JackrabbitNodeTypeManager.

Capitolo 5

Realizzazione della Gestione Documentale

Galileo ERP [4] è una suite applicativa moderna, sviluppata in Italia che ha l'ambizione di essere snella, completa, sicura, modulare e aperta, supporto ideale per l'informatizzazione di molte tipologie di aziende di produzione e commerciali.
Galileo ERP è una applicazione ottima anche per le aziende che puntano a visibilità e organizzazioni internazionali e che per questo necessitano di lavorare con software in lingua estera e con una specifica fiscalità.

Oltre duemila aziende, appartenenti a differenti settori di merceologia e di diversa dimensione, hanno scelto Galileo [24] per le stesse ragioni: rapporto qualità/prezzo, diffusione internazionale e cura nel servizio.
Galileo è la suite verticale che si rivolge alle aziende manifatturiere con l'intento di supportarle nel pieno e completo controllo delle risorse e dei processi gestionali. Con Galileo le imprese possono agevolmente lavorare con il software in lingua gestendo la specifica fiscalità in decine di paesi tra Europa, Americhe, Australia, Russia e Cina.

La suite è una soluzione ERP fra le più diffuse nel mercato (seconda a livello nazionale in AS/400) e si compone di una serie di moduli applicativi rivolti a specifiche funzioni aziendali. Grazie a questa modularità permette l'implementazione di una soluzione configurata sulla base delle reali esigenze ed i moduli principali che compongono Galileo sono dedicati alla gestione dell'area amministrativa, di quella commerciale e di quella produttiva.

L'alta specializzazione permette a Galileo di avvalersi anche di specifiche soluzioni che permettono la riorganizzazione dell'intero processo produttivo, inclusi gli aspetti inerenti all'ottimizzazione della logistica e all'organizzazione aziendale.

Nella versione completa di Galileo si trovano i seguenti moduli:

- Amministrazione
- Gestione Commerciale
- Produzione
- Controllo di Gestione

- Assicurazione Qualità
- Business Intelligence
- Configurazione Prodotti
- CRM
- Gestione Documentale
- Schedulazione
- Gestione della Fabbrica
- Assistenza e Post Vendita
- Project Management
- Key Performance Indicators
- Supply Chain Management

Si è brevemente analizzata la gestione documentale presente in Galileo per poter implementare, in parallelo, la gestione documentale con Java Content Repository.
Per prima cosa per la realizzazione della nuova gestione documenti sono state inserite le librerie:

- jcr-1.0.jar (specifiche JSR-170) [1]
- jackrabbit-jca-1.6.0.jar (specifiche di Apache Jackrabbit) [2]
- jackrabbit-jcr-rmi-1.5.0.jar (specifiche di Apache Jackrabbit per il Content Repository in remoto) [3]
- jdom.jar (specifiche per la gestione dei file xml) [4]

in Eclipse, ambiente di sviluppo utilizzato per creare le classi Java che andranno a realizzare la gestione documentale con Java Content Repository.
Sempre in Eclipse, inoltre, è stato inserito il plugin JCRBrowser [5] per l'analisi del contenuto interno del repository, come lo storico delle versioni e le registrazioni dei nuovi tipi di nodo.

In seguito, è stato configurato un server in una macchina diversa da quella utilizzata per sviluppare la gestione documenti. In questo server è stato installato

[1] http://jcp.org/en/jsr/detail?id=170
[2] http://www.apache.org/dyn/closer.cgi/jackrabbit/1.6.1/jackrabbit-jca-1.6.1.rar
[3] http://jackrabbit.apache.org/commons/jcr-rmi/1.5.html
[4] http://www.jdom.org/
[5] http://sourceforge.net/projects/jcrbrowser/

Apache Tomcat [6] ed effettuato il deploy di jackrabbit.war, contenente il necessario per far funzionare Apache Jackrabbit. Automaticamente si è creata la cartella jackrabbit con all'interno due file: repository.xml e bootstrap.properties e in più le cartelle log, repository, tmp, version e workspaces.
Dopo aver sistemato il file repository.xml come già analizzato nel Capitolo 4 è stato sistemato anche il file bootstrap.properties inserendo i corretti percorsi delle cartelle, la porta rmi.port da 0 a 1099, l'indirizzo rmi.host da localhost a ip-server e aggiunta la voce rmi.uri completa di tutto l'indirizzo ip-server:porta.

Quanto in seguito esposto è stato sviluppato in Eclipse, compilato ed eseguito con Java 5 [7].

Ogni sezione rappresenta una parte della gestione documentale.
Tutte le immagini proposte mostrano il JCR sviluppato con dati non corrispondenti al vero, in quanto non è stato possibile riportare i dati aziendali perché rappresentavano dati sensibili.

Un'applicazione utilizzata per gestire i documenti deve essere in grado di:

- Inserire e rimuovere documenti
- Modificare, aggiungere o rimuovere proprietà
- Creare revisioni del documento
- Interrogare il repository e ricevere il documento corretto

Si fa uso in ogni classe di:

Repository repository = new URLRemoteRepository("http://ip-server:porta/jackrabbit/rmi");

per ottenere l'oggetto Repository al quale poi richiedere la sessione con:

Session session = repository.login("username" , "password".toCharArray());

E' possibile entrare nel repository con un qualsiasi utente avente tutti i privilegi. La gestione dell'accesso rappresenterà un lavoro futuro.

5.1 Creazione di nodi personalizzati

Nella gestione documentale presente in Galileo non era possibile inserire delle nuove proprietà per un solo documento, ovvero se un documento rappresentava una fattura le proprietà destinate ad essa erano già prestabilite e non modificabili. Con la gestione realizzata mediante un Content Repository è stato possibile eliminare questo limite grazie alla possibilità di personalizzare i nodi, soprattutto di creare nuove proprietà in tempi ragionevoli.

[6] http://tomcat.apache.org/
[7] http://java.sun.com

Dopo aver analizzato la struttura della gestione documenti di Galileo si sono scelte cinque proprietà da rendere obbligatorie in ogni documento e una proprietà * la quale può contenere un qualsiasi numero di altre proprietà di tipo non definito a priori.

Il nuovo nodo è stato definito in un file con estensione .cnd, Compact Namespace and Node Type Definition, come illustrato nel Codice 5.1.

```
<mix = ‘‘http://www.jcp.org/jcr/mix/1.0’’>
<nt = ‘‘http://www.jcp.org/jcr/nt/1.0’’>
<personalize=‘‘http://ip−server:port/repository/default/personalize’’ >

[personalize: file ] > nt: file

[personalize:resource] > nt:resource

[personalize:document] > personalize:resource
− Ditta          (String) mandatory
− TipoDocumento  (String) mandatory
− Anno           (String) mandatory
− IdDocumento    (String) mandatory
− PathDocumento  (String) mandatory
− *              (*)
```

Codice 5.1: File CND di definizione del nuovo nodo personalizzato.

Quando verrà inserito un documento le proprietà mandatory sopra elencate dovranno essere citate, altrimenti il documento non verrà aggiunto. Inoltre le prime quattro vengono anche utilizzate per dare una struttura all'albero contenuto nel workspace, del tipo directory-sottodirectory, come illustrato in Figura 5.1.

Si è scelto di dare una struttura al JCR per un maggiore ordine visivo, ma soprattutto perchè sarà molto utile per modificare proprietà, creare versioni e ricercare un determinato documento quando il file è già stato immagazzinato. Altrimenti, tutti i documenti sarebbero stati immagazzinati subito dopo il nodo root.

Per poter usare il nodo personalizzato è necessario registrare il namespace, in questo caso 'personalize', e registrare il nodo definito nel file cnd. Quindi dopo aver ottenuto una sessione viene registrato il namespace in questo modo:

session.getWorkspace().getNamespaceRegistry().registerNamespace(“personalize”, “http://ip-server:port/repository/default/personalize”);

sarà possibile solo da questo momento in poi registrare il nodo. Sempre dal Workspace si ricava il *NodeTypeManager* il quale ritorna un oggetto di tipo *ClientJackrabbitNodeTypeManager*.
Ottenuto il manager, con il metodo

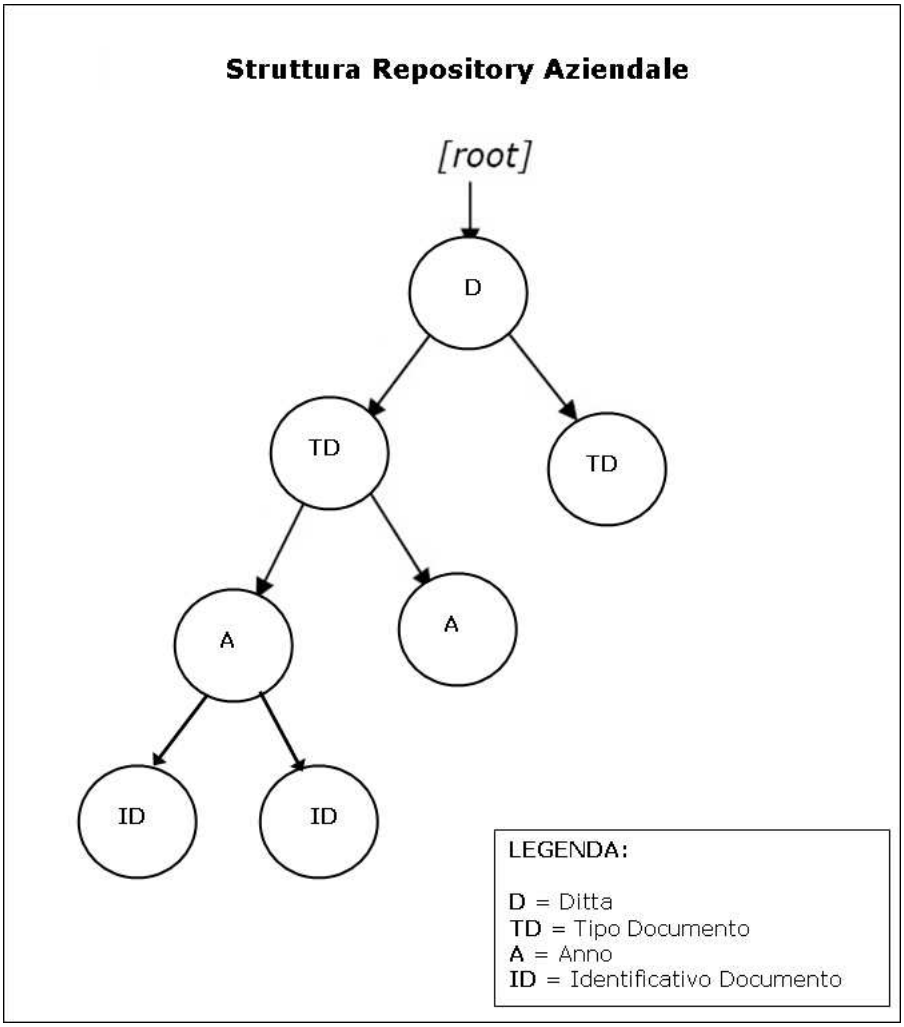

Figura 5.1: Struttura JCR Aziendale.

registerNodeType(new FileInputStream(cndFileName),
ClientJackrabbitNodeTypeManager.TEXT_X_JCR_CND);

il quale richiede il nome del file e il formato di definizione, il nodo verrà registrato e sarà pronto per essere usato.

A questo punto il nodo sarà presente in

/jcr:system/jcr:NodeTypes

inoltre nella cartella jackrabbit/repository/namespace verrà creato il file.properties con l'aggiunta del nuovo namespace e in jackrabbit/repository/nodetype ci sarà un nuovo file xml custom_nodetypes con all'interno quanto definito nel file cnd. Come si può vedere dalla Figura 5.2 la freccia indica il nuovo nodo inserito, questo sarà visibile soltando dal JCRBrowser, quindi solo dal 'lato sviluppatore'.

Figura 5.2: Inserimento di un nodo personalizzato.

5.2 Inserimento di nuovi documenti

Per il momento è stato creato un parallelismo tra la gestione documenti di Galileo e la gestione documenti creata con il Content Repository. Ovviamente così non potranno essere sfruttare a pieno tutte le funzionalità del JCR ma risulta essere una buona partenza per poter testare le sue potenzialità.

Quando Galileo inserisce un documento, in parallelo crea un file xml contenente tutte le proprietà e invocherà i metodi per l'inserimento, così per il momento il documento sarà immagazzinato sia in Galileo che nel JCR.

Il file xml conterrà obbligatoriamente le proprietà: Ditta, TipoDocumento, Anno, IdDocumento, PathDocumento (tutte di tipo String). Potranno essere inseriti altri attributi ma la struttura di ciascuna proprietà, <*ITEM*>, deve rispettare quanto illustrato nel Codice 5.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<LIST>
<ITEM tipo="String" nome="nome_proprieta" valore="valore_proprieta" />
<ITEM tipo="Double" nome="nome_proprieta" valore="valore_proprieta" />
<ITEM tipo="Date" nome="nome_proprieta" valore="valore_proprieta" />
<ITEM tipo="Boolean" nome="nome_proprieta" valore="valore_proprieta" />
</LIST>
```

Codice 5.2: XML per l'inserimento del documento.

Quando il file xml è stato creato, Galileo stesso chiamerà il metodo per l'inserimento dei documenti, questo per prima cosa legge il contenuto del file xml e inserisce le proprietà in base al tipo nelle rispettive HashMap. Quando tutte le proprietà saranno inserite nelle corrispondenti HashMap si cercherà di ottenere una sessione e verrà immagazzinato il documento. Come già brevemente spiegato prima, nella root verrà inserito il nodo contenente il documento.
Il nodo Ditta, all'interno di questo, il nodo TipoDocumento e ancora il nodo Anno, tutti di tipo *nt:unstructured* e soltanto il nodo IdDocumento, a sua volta dentro al nodo Anno, sarà di tipo *personalize:file*. Si è scelto così, perchè i nodi Ditta, TipoDocumento e Anno potrebbero essere riutilizzati, nel senso che una specifica ditta potrà avere tanti documenti di un determinato tipo in un certo anno, quindi sarà IdDocumento il nodo di riferimento per l'occhio umano. Una volta inseriti i tre nodi, quando si riproporranno, basterà iterarli e se già presenti inserire soltanto il nodo riguardante IdDocumento. Per inserire il nodo allora si avrà:

Node ditta = root.addNode(hm.get("Ditta").toString(), "nt:unstructured");
Node tipo =
ditta.addNode(hm.get("TipoDocumento").toString(), "nt:unstructured");
Node anno = tipo.addNode(hm.get("Anno").toString(), "nt:unstructured");
Node file =
anno.addNode(hm.get("IdDocumento").toString(), "personalize:file");

Per inserire un nodo quando i primi tre nodi sono già stati creati basterà scrivere:

Node file = parent.getNode(hm.get("Ditta").toString())
.getNode(hm.get("TipoDocumento").toString())
.getNode(hm.get("Anno").toString())
.addNode(hm.get("IdDocumento").toString(), "personalize:file");

Tutto questo prepara il nodo file per inserire il content, ovvero il contenuto binario del documento, con:

Node content = file.addNode("jcr:content", "personalize:document");

verranno inserite tutte le proprietà compreso il documento stesso.

Per settare le proprietà sarà necessario iterare le mappe e poi invocare:

Figura 5.3: JCR iniziale (lato sviluppatore).

content.setProperty((String) me.getKey(), (...) me.getValue());

me rappresenta una singola coppia chiave, valore della mappa, quindi

me.getKey()

rappresenta il nome della proprietà e

me.getValue()

i valori, dove (...) potrà contenere uno tra i tipi disponibili: String, Double, Boolean, Date. Inoltre, visto che *personalize:document* eredita le proprietà da *nt:resource* si avranno in più anche le proprietà:

content.setProperty("jcr:mimeType", mimeType);

dove mimeType rappresenta l'estensione del file,

content.setProperty("jcr:encoding", "");

la codifica,

content.setProperty("jcr:lastModified", Calendar.getInstance());

l'ora e il giorno dell'ultima modifica e

content.setProperty("jcr:data", new FileInputStream((String) content.getProperty("PathDocumento").getValue().getString()));

questa è la proprietà che immagazzina il file in binario all'interno del repository. E' necessario effettuare un salvataggio altrimenti tutte le modifiche essendo volatili andranno perse.
La Figura 5.3 mostra il JCR prima dell'inserimento del documento e la Figura 5.5 illustra il contenuto del JCR dopo l'inserimento del documento, dal lato sviluppatore, JCRBrowser.
Le Figure 5.4 e 5.6, invece, illustrano il contenuto del JCR prima e dopo l'inserimento del documento visibile al cliente da un qualsiasi browser Web.
Inoltre nella Figura 5.7 sono illustrate le proprietà del nodo file, il quale è di tipo personalize:file, e nella Figura 5.8 le proprietà del nodo content, il quale è di tipo personalize:document, visibili nella struttura presentata dal lato sviluppatore, ricercabili, in quando indicizzate di default, dal client nel browser Web.

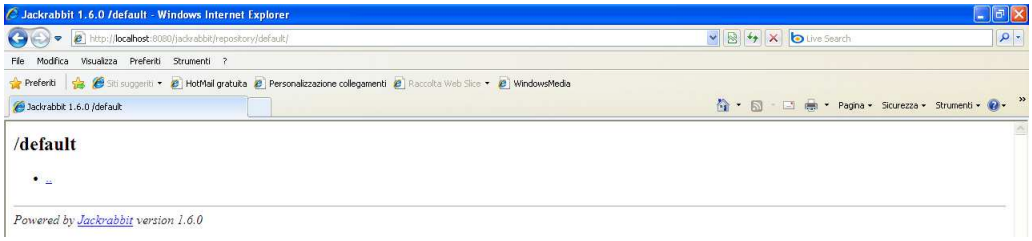

Figura 5.4: JCR iniziale (lato client).

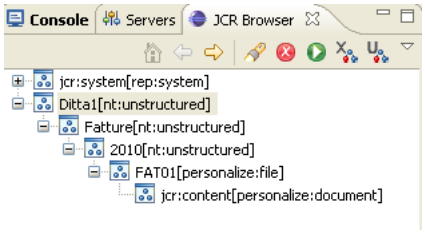

Figura 5.5: Inserimento di un nodo contenente un documento (lato sviluppatore).

5.3 Rimozione di un documento

Per rimuovere un documento si agirà come per gli inserimenti, Galileo nel momento in cui decide di eliminare il documento crea il file xml con le quattro proprietà che permettono di iterare i nodi ovvero Ditta, TipoDocumento, Anno, IdDocumento, queste verranno inserite in un'unica HashMap in quanto rappresentano tutte Stringhe.

$$\begin{aligned} &\textit{Node file} = \textit{parent.getNode(hm.get(Ditta).toString())} \\ &\textit{.getNode(hm.get(TipoDocumento).toString())} \\ &\textit{.getNode(hm.get(Anno).toString())} \\ &\textit{.getNode(hms.get(IdDocumento).toString());} \end{aligned}$$

Ottenuto così il documento, chiamando il metodo

$$\textit{remove()}$$

e salvando le modifiche il documento verrà rimosso.

5.4 Inserimento di una nuova proprietà

Come già accennato, in Galileo risulta difficile creare singoli nuovi attributi in singoli record, o meglio si potrebbero creare dei record dinamici, ma questo è molto sconsigliato in quanto le prestazioni diventerebbero scarsissime. Quindi verrà comunque usato Galileo per l'inserimento di una nuova proprietà, ma questo verrà effettuato soltanto nel JCR. Sarà un test, perchè lavorando parallelamente ci si limita, per il momento, a ciò che Galileo permette di fare; quando il JCR verrà utilizzato autonomamente l'inserimento di una nuova proprietà in un singolo

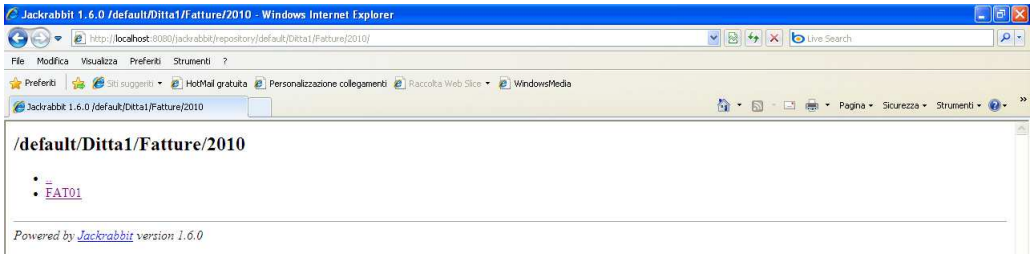

Figura 5.6: Inserimento di un nodo contenente un documento (lato client).

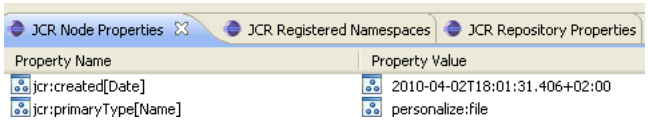

Figura 5.7: Proprietà del nodo file.

nodo verrà utilizzata nella sua completezza. Ovviamente l'obiettivo era quello di capire tutte le possibili funzionalità del JCR.

Galileo andrà a creare un file xml con le proprietà Ditta, TipoDocumento, Anno, IdDocumento e in aggiunta nello stesso formato già presentato nel Codice 5.2 ci saranno degli ulteriori $<ITEM>$ contenenti le nuove proprietà. Sempre Galileo una volta creato il file xml invocherà il metodo di inserimento delle proprietà, il quale prenderà tutti i valori del file xml e li inserirà nelle corrispondenti HashMap. Avendo le quattro proprietà il JCR ricava il documento con IdDocumento richiesto e analizza tutte le proprietà in esso immagazzinate. Se le nuove proprietà non sono già contenute nel documento allora mediante il

$$setProperty(nameProperty,\ valueProperty)$$

verrà aggiornato il repository. Con un salvataggio saranno rese permanenti le modifiche.
Partendo dalla Figura 5.8 si può notare che la Figura 5.9 si differenzia nell'inserimento della nuova proprietà Agente.

5.5 Modifica di una proprietà esistente

Anche per modificare delle proprietà esistenti, il comportamento è identico all'inserimento di una nuova proprietà. Se nel momento in cui vengono iterate tutte le proprietà presenti, si trova quella da modificare, si immagazzina soltanto il nuovo valore corrispondente al nome della proprietà.
La Figura 5.10 illustra la modifica al valore della proprietà Agente.

5.6 Rimozione di una proprietà esistente

Per rimuovere delle proprietà esistenti Galileo produrrà sempre un file xml leggermente diverso da quelli precedentemente esposti. Oltre alle solite quattro

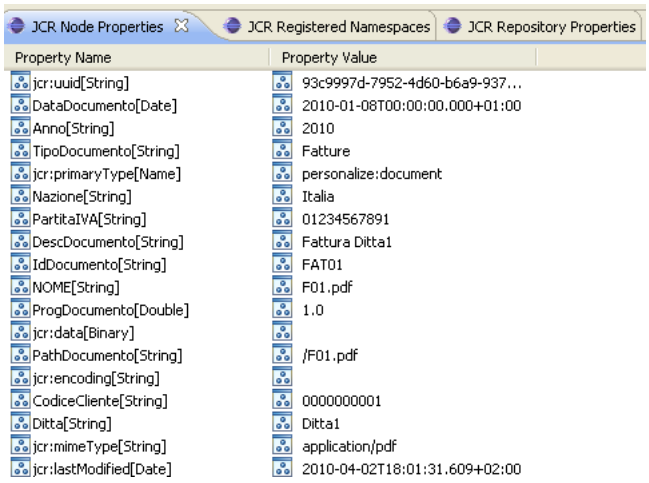

Figura 5.8: Proprietà del nodo content.

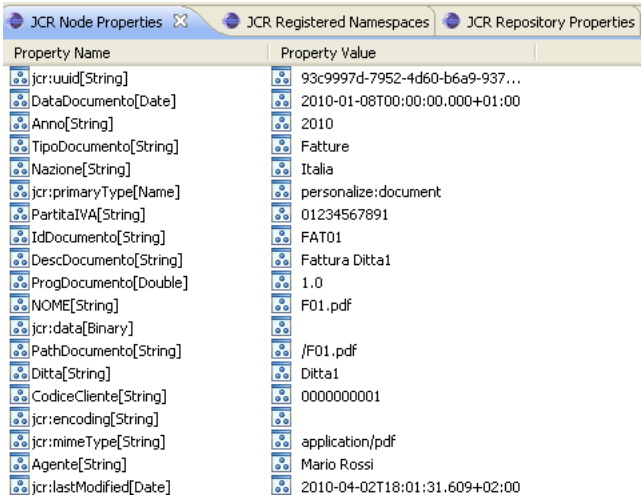

Figura 5.9: Inserimento di una nuova proprietà in un singolo nodo già archiviato.

proprietà esistenti aggiungerà uno o più <*ITEM*>, come illustrato nel Codice 5.3.

```
1 <ITEM>tipo=“RIMUOVI” nome=“nome_proprieta” valore=“”</ITEM>
```

Codice 5.3: XML per la rimozione di una proprietà.

L'elemento tipo non conterrà più il tipo di dato ma il valore RIMUOVI e verrà compilato soltanto l'attributo nome con il rispettivo nome della proprietà da eliminare. Dopo aver creato il file xml delle proprietà, Galileo chiama il metodo per la rimozione delle proprietà. Immagazzina nella HashMap delle Stringhe le quattro proprietà e nella HashMap della rimozione i nomi delle proprietà che hanno come tipo RIMUOVI. Una volta ottenuto il documento, verranno iterate

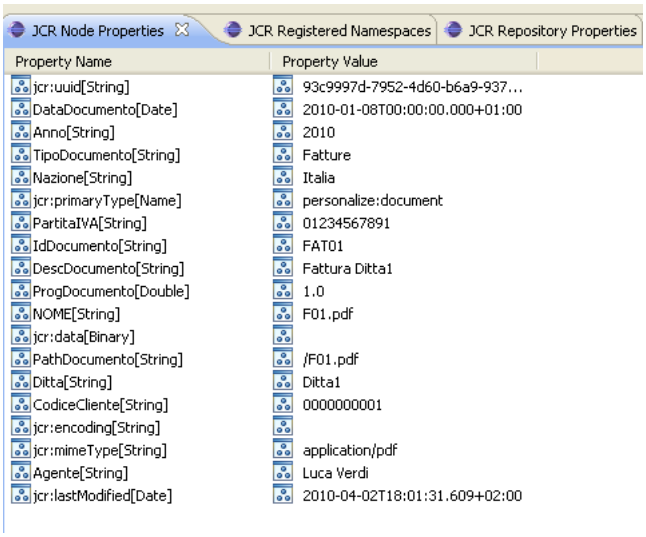

Figura 5.10: Modifica di una proprietà già presente in un singolo nodo.

tutte le proprietà e quando il nome della proprietà del documento sarà uguale a quella contenuta nella mappa della rimozione, mediante l'uso di

content.getProperty((String)hm.getKey()).remove();

verranno rimosse una per volta le proprietà richieste.

5.7 Versioni

Come già illustrato nei precedenti capitoli le versioni sono delle revisioni dei documenti. E' possibile che si verifichino due casi: o già si sa che un documento avrà delle revisioni quindi verrà aggiunto il mix:versionable nel momento in cui si carica il documento nel repository oppure il documento è già presente nel repository e il nodo che lo contiene deve essere reso versionable in seguito.
Verranno ora analizzati i due casi.

1. Nel primo caso l'utente che inserirà il documento è a conoscenza che il file che andrà ad inserire conterrà delle versioni, allora utilizzerà il comando di Galileo che permette l'inserimento di un documento revisionabile (la gestione documenti già presente in azienda permette di inserire documenti che supportano versioni). Quando verrà attivato il comando di Galileo, automaticamente verrà prodotto il file xml come per l'inserimento di un documento, ma verrà in seguito invocato il metodo per l'inserimento di un nodo contenente file con versioni. A differenza del normale inserimento questo metodo contiene l'aggiunta del tipo di nodo *mix:versionable* quindi verranno invocati:

file.addMixin(JcrConstants.MIX_VERSIONABLE);
e
content.addMixin(JcrConstants.MIX_VERSIONABLE);

sia il file che il suo contenuto vengono resi versionable.

Quando il documento sarà inserito nel JCR, subito dopo aver salvato le modifiche, è necessario invocare

file.checkin()
e
content.checkin()

con questi due comandi viene archiviata la prima versione del documento, quella corrente.

2. Nel secondo caso l'utente si renderà conto che dovrà inserire una versione ma il documento che aveva in precedenza immagazzinato non gli consente di inserire una nuova revisione. Questo fa comprendere che il nodo non era stato reso versionable all'inizio. Allora utilizzando il comando appropriato da Galileo verrà prodotto il file xml contenente le quattro proprietà che permettono di ripescare il documento e si dichiarerà il nodo ottenuto *mix:versionable* come già esposto per il primo caso.

Quando si ha un nodo versionable è possibile aggiungere una nuova versione. Per versione o revisione si intende una correzione, un controllo o semplicemente delle aggiunte ad un documento presente.

Galileo produrrà il file xml con le cinque proprietà, la quinta proprietà sarà il path del nuovo file da inserire. Il metodo che permetterà di aggiungere una nuova versione sarà simile a quello per l'aggiunta di un documento. Dopo aver trovato il giusto documento da revisionare lo si dovrà aprire con

file.checkout()
e
content.checkout()

si avrà quindi la possibilità di inserire una nuova versione, verranno settate delle nuove proprietà, se ci sono, o solamente registrate le quattro proprietà obbligatorie del nodo *nt:resource*. Quindi *jcr:mimeType*, *jcr:encoding*, *jcr:lastModified*, *jcr:data* ovviamente quest'ultima dovrà contenere il percorso con il nuovo file da caricare. Dopo il salvataggio è necessario invocare checkin().
La Figura 5.11 illustra l'inserimento di un nodo contenente un documento che potrà essere revisionato.
Nelle Figure 5.12 e 5.13 vengono rispettivamente mostrate le proprietà dei nodi file e content, nel momento in cui vengono dichiarati mix:versionable.

5.7.1 Iterare lo storico delle versioni

E' possibile iterare lo storico delle versioni e ottenere così tutte le revisioni presenti al suo interno. Per fare ciò Galileo produrrà il file xml con le quattro proprietà del file da cercare. Ottenuto il file si invocheranno i metodi

getVersionHistory()

il quale restituirà lo storico e

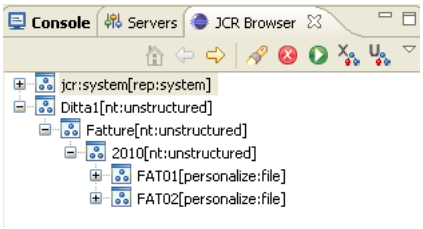

Figura 5.11: Inserimento di un nodo che ammette versioni.

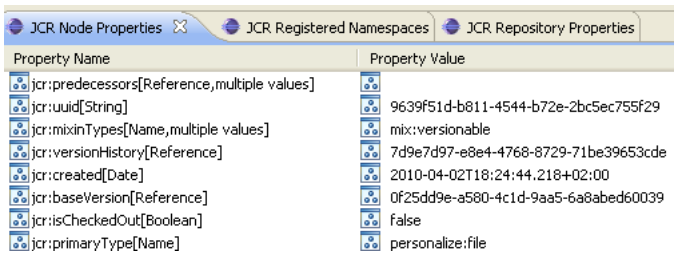

Figura 5.12: Struttura del nodo file reso mix:versionable.

getAllVersions()

il quale permette di ottenere l'iteratore delle versioni del documento. Iterando l'iteratore delle versioni si ottengono le proprietà di ciascuna revisione. Queste vengono scritte in un file xml dei risultati (Codice 5.4) e passato a Galileo, il quale a sua volta leggerà l'xml dei risultati e produrrà a video l'ordine delle versioni, così l'utente deciderà mediante un semplice click quale documento andare a vedere. Le Figure 5.14 e 5.15 mostrano lo storico delle versioni dal lato sviluppatore, la prima illustra il primo documento inserito, il quale può essere revisionato, la seconda immagine mostra una nuova versione del documento.

```
<RISULTATI Risultato="numero versioni">
<VALORI>
<VERSIONE>Numero versione</VERSIONE>
<TipoDocumento>Tipologia documento</TipoDocumento>
<PathFile>Path del file</PathFile>
<Ditta>ditta</Ditta>
<Anno>Anno</Anno>
<Nome>Nome file</Nome>
<IdDocumento>Identificativo</IdDocumento>
</VALORI>
```

Codice 5.4: XML dei risultati dell'iterazione delle versioni.

5.7.2 Ritornare una specifica versione

Quando l'utente selezionerà il documento da vedere, Galileo produrrà il file xml con le proprietà del documento scelto compreso il numero della versione, quest'ul-

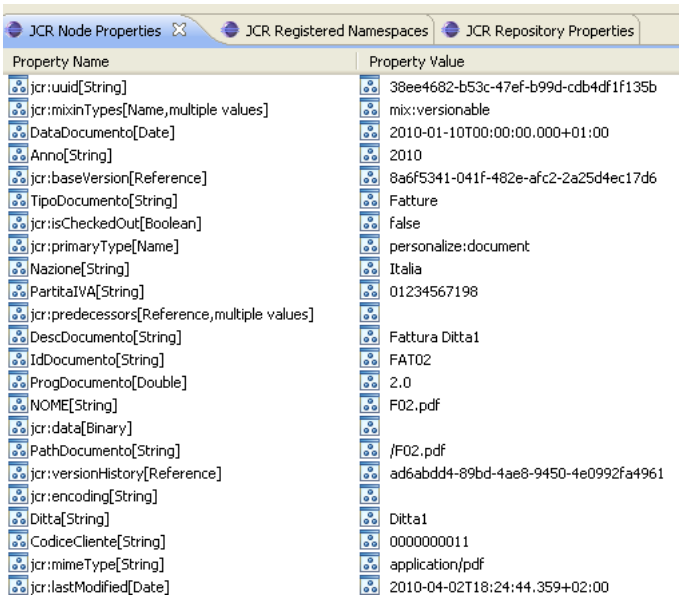

Figura 5.13: Struttura del nodo content reso mix:versionable.

timo scritto con il tag presentato nel Codice 5.5, lo passerà al JCR, il quale inserirà tutto nelle rispettive HashMap e nuovamente ricercherà il nodo ed effettuerà l'iterazione dello storico.

```
<ITEM Tipo=“tipo di dato” nome=“VERSIONE” valore=“numero versione”>
```

Codice 5.5: XML per richiedere una specifica versione.

Quando il nome della versione sarà uguale al valore della versione passata dal file xml verrà prodotto un file di uscita avente il nome del file con l'aggiunta del tempo in millisecondi.
Nel Codice Java 5.6 viene illustrata la parte di codice che si occupa di quanto spiegato in precedenza.

Il file prodotto dovrà essere inviato nel path di uscita che Galileo aveva inserito nella firma del metodo invocato per ottenere la versione.

Da tenere presente che il numero della versione inserito all'interno dell'xml deve essere completo, così scritto 1.numero progressivo della versione.
Ad esempio 1.0 rappresenta la prima versione, 1.1 la seconda, 1.2 la terza e così via.

5.8 Query

Per interrogare il JCR e quindi ottenere i documenti in esso contenuti verrà aperta da Galileo la sua l'interfaccia grafica delle query. Verranno compilate, in base alla ricerca da effettuare, le label con dei valori e nel momento in cui verrà premuto l'invio, Galileo preparerà il file xml con i valori inseriti.
La struttura del file prodotto deve essere come presentata nel Codice 5.7.

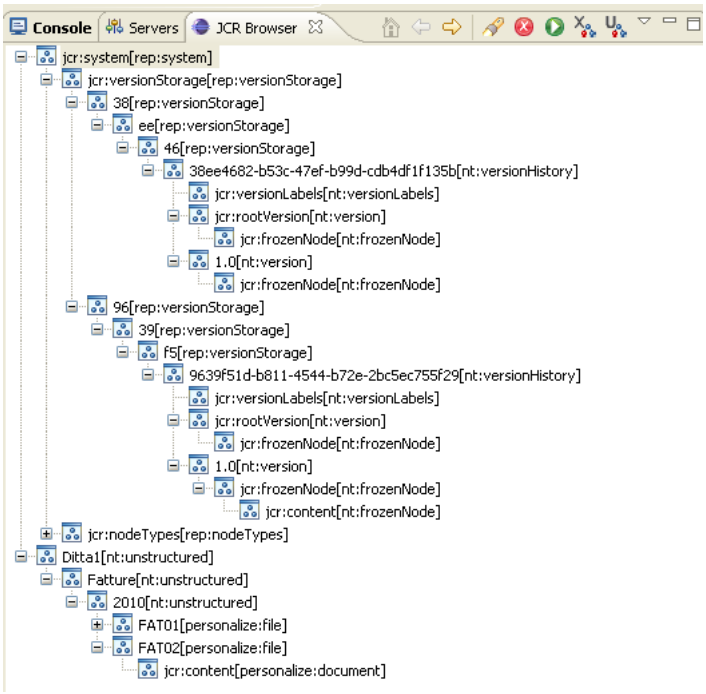

Figura 5.14: Struttura dello storico delle versioni.

Se gli elementi contenuti nel tag <*VALORI*> sono maggiori di uno, saranno separati dal simbolo '|'; è stato scelto questo carattere perchè è l'unico che non viene mai usato da Galileo, quindi c'è la sicurezza che non sarà generata confusione.

Quando il JCR riceve il file xml, verrà letto il contenuto, e di <*ITEM*> in <*ITEM*> si andrà a comporre la stringa da usare poi nella query, negli esempi seguenti tale stringa verrà chiamata xpath.
Quando si incontra tipo uguale a Order o a Content verranno preparate due ulteriori stringhe separate, una per gli ordinamenti (stringorder) e una per i contenuti (stringcontent).

Se tipo, invece, rappresenta un tipo di dato, allora, se il tag <*VALORI*> contiene un numero di valori maggiori di uno, questi verranno scomposti e il nome della proprietà con i rispettivi valori saranno inseriti nella HashMap in base al tipo di dato. Se il valore è unico verrà subito inserito nella HashMap corrispondente.

I valori delle proprietà contenuti nella HashMap verranno temporaneamente trasferiti in un array per poter creare più agevolmente la stringa. Quando i valori stanno nelle HashMap verrà composta pezzo per pezzo la stringa. In seguito saranno esposti tutti i casi con la creazione delle rispettive stringhe. Quando tutto il file xml sarà stato letto e quindi tutte le stringhe composte sarà possibile assemblarle e creare la stringa finale in sintassi XPath che permetterà di ottenere i risultati della query.

Operatore di confronto tra due valori compresi: BETWEEN
<ITEM tipo="tipo dato" nome="nome_proprietà" operatore="BETWEEN">

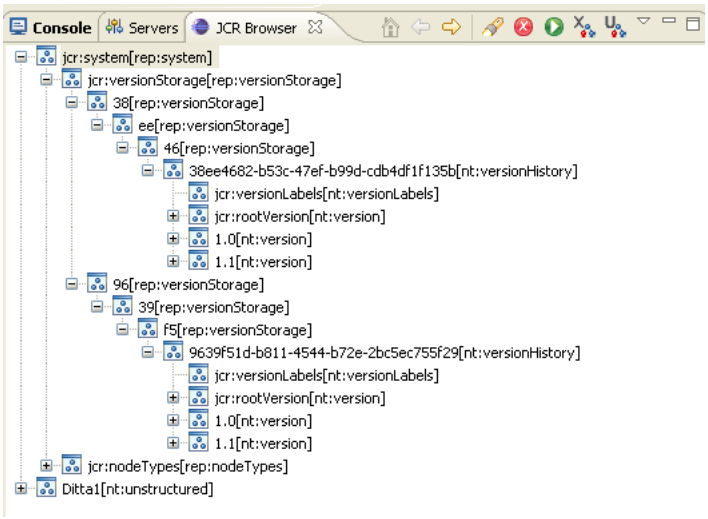

Figura 5.15: Inserimento di una nuova versione.

```
if(version.getName().equals(hm.get("VERSIONE").toString())){
InputStream stream = (session.getNodeByUUID(currentNode.getUUID())
.getNode("jcr:content").getProperty("jcr:data")).getStream());
Date da = new Date(System.currentTimeMillis());
path=pathout+""+da+"_"+da.getTime()+"_"+currentNode
.getNode("jcr:content").getProperty("Nome").getValue().getString();
FileOutputStream fos = new FileOutputStream(path);
byte[] contentbyte = new byte[stream.available()];
stream.read(contentbyte);
fos.write(contentbyte);
fos.flush();
fos.close();
}
```

Codice 5.6: Codice Java per ritornare una specifica versione.

<VALORI>primoValore|secondoValore</VALORI>
</ITEM>

xpath=xpath+"@"+(String)me.getKey()+">="+arrayValori[i]+"and @"+(String)me.getKey()+"<="+arrayValori[i+1]+" and ";

Operatore di uguaglianza: =
<ITEM tipo="tipo dato" nome="nome_proprietà" operatore="=">
<VALORI>valore</VALORI>
</ITEM>

xpath=xpath+"@"+(String)me.getKey()+"="+arrayValori[i]+" and ";

```
<ITEM Tipo="Tipo dato" nome="nome_proprietà" Operatore="Operatore">
<VALORI>Valori</VALORI>
</ITEM>
```

Codice 5.7: Struttura generale del file XML per le Query.

Operatore di disuguaglianza: !=
<ITEM tipo=“tipo dato” nome=“nome_proprietà” operatore=“!=”>
<VALORI>valore</VALORI>
</ITEM>

xpath=xpath+“@”+(String)me.getKey()+“!=”+arrayValori[i]+“ and ”;

Operatore di maggioranza stretta: >
<ITEM tipo=“tipo dato” nome=“nome_proprietà” operatore=“MAX”>
<VALORI>valore</VALORI>
</ITEM>

xpath=xpath+“@”+(String)me.getKey()+“>”+arrayValori[i]+“ and ”;

Operatore di minoranza stretta: <
<ITEM tipo=“tipo dato” nome=“nome_proprietà” operatore=“MIN”>
<VALORI>valore</VALORI>
</ITEM>

xpath=xpath+“@”+(String)me.getKey()“<”+arrayValori[i]+“ and ”;

Operatore di minoranza: <=
<ITEM tipo=“tipo dato” nome=“nome_proprietà” operatore=“MIN=”>
<VALORI>valore</VALORI>
</ITEM>

xpath=xpath+“@”+(String)me.getKey()+“<=”+arrayValori[i]+“ and ”;

Operatore di maggioranza: >=
<ITEM tipo=“tipo dato” nome=“nome_proprietà” operatore=“MAX=”>
<VALORI>valore</VALORI>
</ITEM>

xpath=xpath+“@”+(String)me.getKey()+“>=”+arrayValori[i]+“ and ”;

Operatore di maggioranza e minoranza stretta: > <
<ITEM tipo=“tipo dato” nome=“nome_proprietà” operatore=“MINMAX”>
<VALORI>primoValore|secondoValore</VALORI>
</ITEM>

xpath=xpath+“@”+(String)me.getKey()+“>”+arrayValori[i]+“ and @”+(String)me.getKey()+“<=”+arrayValori[i+1]+“ and ”;

Operatore IN
<ITEM tipo=“tipo dato” nome=“nome_proprietà” operatore=“IN”>
<VALORI>primoValore|secondoValore|terzoValore</VALORI>
</ITEM>

in=in+“@”+(String)me.getKey()+“=”+arrayValori[i-n]+“ or ”;
String strin = in.substring(0, in.length()-3);

xpath=xpath+“(”+strin+“)”+“and ”;

Operatore NOT IN
<ITEM tipo=“tipo dato” nome=“nome_proprietà” operatore=“NOT IN”>
<VALORI>primoValore|secondoValore|terzoValore</VALORI>
</ITEM>

not=not+“@”+(String)me.getKey()+“!=”+arrayValori[i-n]+“ and ”;
String strnot = not.substring(0, not.length()-4);

xpath=xpath+“(”+strnot+“)”+“and ”;

Operatore LIKE
<ITEM tipo=“tipo dato” nome=“nome_proprietà” operatore=“LIKE”>
<VALORI>valore da cercare</VALORI>
</ITEM>

Se si conosce una parte del contenuto basta aggiungere prima, dopo o nel mezzo del valore da cercare il simbolo %. Il LIKE analizza solo dati di tipo String.

xpath=xpath+“jcr:like(@”+(String)me.getKey()+“,”+arrayValori[i]+“) and ”;

La stringa finale composta dall’assemblaggio di tutte le stringhe sarà:

stringfinal=(xpath).substring(0, xpath.length()-4);

si tolgono quattro caratteri per eliminare l’ultimo and pendente.

Per gli ordinamenti e il content si avrà:
Operatore CRESCENTE
<ITEM tipo=“Order” nome=“” operatore=“CRESCENTE”>
<VALORI>nomi proprietà da ordinare</VALORI>
</ITEM>

Operatore DECRESCENTE
<ITEM tipo=“Order” nome=“” operatore=“DECRESCENTE”>
<VALORI>nomi proprietà da ordinare</VALORI>
</ITEM>

stringaorder=stringaorder+"@"+proprieta[i]+" ascending, ";
orderfinal = "order by "+stringaorder.substring(0, stringaorder.length()-2);

Nel caso in cui l'ordinamento sia descrescente ci sarà una netta somiglianza all'ordinamento crescente, cambierà soltando ascending con descending.

Operatore CONTENT
<ITEM tipo="Content" nome="nome_proprietà" operatore="CONTENT">
<VALORI>contentuto del documento da cercare</VALORI>
</ITEM>

contentfinal = "jcr:contains(., "+stringacontent+")";

dove stringacontent rappresenta la stringa contenuta nel tag <VALORI>.

Si ottengono quindi tre stringhe: la stringa finale delle proprietà, la stringa dell'ordinamento e la stringa del content.
Sarà possibile invocare il metodo per l'esecuzione della query.
Si potrà avere la presenza o l'assenza della stringa contentfinal, quindi sarà presente un controllo sulla lunghezza della stringa, se la lunghezza sarà maggiore di zero allora la query diventerà:

queryString = "//element(, nt:resource)["+contentfinal+"*
and"+stringfinal+"]"+orderfinal;

altrimenti sarà:

queryString = "//element(, nt:resource)["+stringfinal+"]"+orderfinal;*

Tale ricerca analizzerà tutti i nodi di tipo nt:resource o nodi che ereditano come supertipo *nt:resource*, ad esempio il nodo creato è di tipo *personalize:document* figlio di nt:resource, quindi questa query andrà bene per qualsiasi nodo personalizzato che eredita le proprietà da *nt:resource*.

La query verrà creata con il metodo:

Query query = qm.createQuery(queryString, Query.XPATH);

nella firma del metodo verranno inserite la stringa XPath creata e la costante che indica la sintassi con cui è stata scritta. Verrà eseguita poi con:

QueryResult queryResult = query.execute();

mediante l'iteratore delle righe chiamato con

queryResult.getRows();

```
<?xml version="1.0" encoding="UTF-8"?>
<RISULTATI Totale="numero documenti trovati">
<VALORI>
    <Ditta>valore</Ditta>
    <TipoDocumento>valore</TipoDocumento>
    <Anno>valore</Anno>
    <IdDocumento>valore</IdDocumento>
    <NomeFile>valore</NomeFile>
    <Testo><![CDATA[parte del testo cercato contenuta nel documento,
al testo ricercato verrà assegnato il carattere grassetto]]></Testo>
  </VALORI>
</RISULTATI>
```

Codice 5.8: XML dei risultati delle query.

verrà analizzata ogni singola riga di ciascun risultato.
Come per le versioni, anche per le query verrà costruito un file xml dei risultati, avente struttura presentata nel Codice 5.8.

Galileo quindi leggerà il file xml dei risultati e lo produrrà a video.
L'utente vedrà tutta le serie dei risultati prodotti, potrà quindi analizzarli e se vorrà visualizzare un documento integrale, cioè il file, cliccherà sul rispettivo risultato.
In questo modo, Galileo, nuovamente produrrà un file xml con le quattro proprietà Ditta, TipoDocumento, Anno, IdDocumento, le quali permettono di ricavare il file. Lo invierà al JCR, il quale lo leggerà e ritornerà il documento avente le proprietà richieste, che a sua volta Galileo aprirà e mostrerà all'utente.

Capitolo 6

Conclusioni e sviluppi futuri

L'obiettivo principale del progetto comprendeva lo studio, la progettazione e la realizzazione di un Content Repository per la gestione documentale di Sanmarco Informatica. Le motivazioni principali che spinsero l'azienda ad investire in tale progetto furono principalmente tre: la possibilità di memorizzare dati eterogenei, la possibilità di effettuare ricerche full text in quanto il JCR indicizza tutto il contenuto testuale di un documento ed infine la possibilità di sfruttare una struttura 'dinamica', ovvero un albero di nodi e proprietà.

Come si è potuto vedere nel Capitolo 5, la gestione documentale creata con il Content Repository funziona in parallelo a quella di Galileo. Si è deciso di migrare dal database relazionale con cui è implementato Galileo al Content Repository in maniera graduale per testare passo passo la correttezza del funzionamento.

Al momento attuale in Galileo, grazie all'uso del JCR, si è aggiunta la possibilità di effettuare ricerche full text e immagazzinare singole proprietà in singoli nodi in tempi ottimi.

Il JCR sta 'dietro le quinte' quindi un utente non si renderà conto che utilizzando Galileo in realtà sta lavorando anche con il JCR; Galileo comunque conserva ancora tutte le funzionalità possedute prima del parallelismo con il JCR.

Sono stati mantenuti per il momento i riferimenti presenti in Galileo, per inserirli anche nel JCR sarebbe stato necessario sincronizzare le due gestioni documentali. Quando il JCR sarà reso autonomo verrà analizzato ancor più in dettaglio l'aspetto delle relazioni fra i nodi.

Era desiderio aziendale poter produrre una gestione documenti basata su progetti Open Source e così avvalendosi di Apache Jackrabbit, implementazione di riferimento delle specifiche JSR-170, è stato rispettato tale requisito. Inoltre risulta essere anche multipiattaforma, stabile e adattabile.

In più la gestione documentale creata assume una forma 'generalizzata', cioè tutte le classi che implementano la gestione ricevono da input dei file xml per il passaggio dei parametri. Quindi la gestione documentale potrebbe vivere di vita propria (ovviamente creando i tipi di nodi adatti al caso e scegliendo le opportune proprietà), ad esempio predisponendo delle interfacce grafiche per il passaggio dei parametri, una volta raccolti i dati, il loro invio genererà un file xml letto e usato dal JCR.

Oggi il JCR permette l'accesso a un utente qualsiasi, nei prossimi mesi verrà attuato il meccanismo di sicurezza, già si è scelto che la gestione dell'accesso dovrà essere regolata da Web. Verranno create delle pagine jsp, visibili con il server attivo, per l'autenticazione dei nuovi utenti. Per questo, si utilizzeranno anche le recenti specifiche JSR-283, le quali contengono strumenti utili per la gestione dell'accesso.

Dato che a febbraio 2010 è uscita la versione di Apache Jackrabbit 2.0 la gestione documentale progettata sarà testata anche con tale versione, per vederne la correttezza, anche se già si è a conoscenza che tutte le versioni di Jackrabbit sono retroattive e stabili quindi in linea di massima non dovrebbero esserci enormi cambiamenti.

L'obiettivo futuro finale sarà quello di sganciare il JCR dal parallelismo con Galileo e migrare completamente dal database relazionale al Content Repository unificando in esso tutte le gestioni documentali presenti in azienda.

Appendice A

A.1 Definizione notazione dei tipi di nodi

Le seguenti sezioni forniscono la definizione di ciascun tipo di nodo predefinito. Le definizioni del tipo di nodo hanno la seguente struttura:

NodeTypeName
...
Supertypes
...
IsMixin
...
HasOrderableChildNodes
...
PrimaryItemName
...
ChildNodeDefinition
Name ...
RequiredPrimaryTypes ...
DefaultPrimaryType ...
AutoCreated ...
Mandatory ...
OnParentVersion ...
Protected ...
SameNameSiblings ...
.
. (more ChildNodeDefinitions)
.
PropertyDefinition
Name ...
RequiredType ...
ValueConstraints ...
DefaultValues ...
AutoCreated ...
Mandatory ...
OnParentVersion ...

Protected ...
Multiple ...
.
. (more PropertyDefinitions)
.

A.2 mix:lockable

NodeTypeName
mix:lockable
Supertypes
[]
IsMixin
true
HasOrderableChildNodes
false
PrimaryItemName
null
PropertyDefinition
Name jcr:lockOwner
RequiredType STRING
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory false
OnParentVersion IGNORE
Protected true
Multiple false
PropertyDefinition
Name jcr:lockIsDeep
RequiredType BOOLEAN
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory false
OnParentVersion IGNORE
Protected true
Multiple false

A.3 mix:referenceable

NodeTypeName
mix:referenceable
Supertypes

[]
IsMixin
true
HasOrderableChildNodes
false
PrimaryItemName
null
PropertyDefinition
Name jcr:uuid
RequiredType STRING
ValueConstraints []
DefaultValue null
AutoCreated true
Mandatory true
OnParentVersion INITIALIZE
Protected true
Multiple false

A.4 mix:versionable

NodeTypeName
mix:versionable
Supertypes
mix:referenceable
IsMixin
true
HasOrderableChildNodes
false
PrimaryItemName
null
PropertyDefinition
Name jcr:versionHistory
RequiredType REFERENCE
ValueConstraints [nt:versionHistory]
DefaultValue null
AutoCreated false
Mandatory true
OnParentVersion COPY
Protected true
Multiple false
PropertyDefinition
Name jcr:baseVersion
RequiredType REFERENCE
ValueConstraints [nt:version]

DefaultValues null
AutoCreated false
Mandatory true
OnParentVersion IGNORE
Protected true
Multiple false
PropertyDefinition
Name jcr:isCheckedOut
RequiredType BOOLEAN
ValueConstraints[]
DefaultValues [true]
AutoCreated true
Mandatory true
OnParentVersion IGNORE
Protected true
Multiple false
PropertyDefinition
Name jcr:predecessors
RequiredType REFERENCE
ValueConstraints [nt:version]
DefaultValues null
AutoCreated false
Mandatory true
OnParentVersion COPY
Protected true
Multiple true
PropertyDefinition
Name jcr:mergeFailed
RequiredType REFERENCE
ValueConstraints []
DefaultValue null
AutoCreated false
Mandatory false
OnParentVersion ABORT
Protected true
Multiple true

A.5 nt:base

NodeTypeName
nt:base
Supertypes
[]
IsMixin

false
HasOrderableChildNodes
false
PrimaryItemName
null
PropertyDefinition
Name jcr:primaryType
RequiredType NAME
ValueConstraints []
DefaultValue null
AutoCreated true
Mandatory true
OnParentVersion COMPUTE
Protected true
Multiple false
PropertyDefinition
Name jcr:mixinTypes
RequiredType NAME
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory false
OnParentVersion COMPUTE
Protected true
Multiple true

A.6 nt:unstructured

NodeTypeName
nt:unstructured
Supertypes
nt:base
IsMixin
false
HasOrderableChildNodes
true
PrimaryItemName
null
ChildNodeDefinition
Name *
RequiredPrimaryTypes [nt:base]
DefaultPrimaryType nt:unstructured
AutoCreated false
Mandatory false

OnParentVersion VERSION
Protected false
SameNameSiblings true
PropertyDefinition
Name *
RequiredType UNDEFINED
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory false
OnParentVersion COPY
Protected false
Multiple true
PropertyDefinition
Name *
RequiredType UNDEFINED
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory false
OnParentVersion COPY
Protected false
Multiple false

A.7 nt:hierarchyNode

NodeTypeName
nt:hierarchyNode
Supertypes
nt:base
IsMixin
false
HasOrderableChildNodes
false
PrimaryItemName
null
PropertyDefinition
Name jcr:created
RequiredType DATE
ValueConstraints []
DefaultValue null
AutoCreated true
Mandatory false
OnParentVersion INITIALIZE

Protected true
Multiple false

A.8 nt:file

NodeTypeName
nt:file
Supertypes
nt:hierarchyNode
IsMixin
false
HasOrderableChildNodes
false
PrimaryItemName
jcr:content
ChildNodeDefinition
Name jcr:content
RequiredPrimaryTypes [nt:base]
DefaultPrimaryType null
AutoCreated false
Mandatory true
OnParentVersion COPY
Protected false
SameNameSiblings false

A.9 nt:linkedFile

NodeTypeName
nt:linkedFile
Supertypes
nt:hierarchyNode
IsMixin
false
HasOrderableChildNodes
false
PrimaryItemName
jcr:content
PropertyDefinition
Name jcr:content
RequiredType REFERENCE
ValueConstraints []
DefaultValue null
AutoCreated false
Mandatory true

OnParentVersion COPY
Protected false
Multiple false

A.10 nt:folder

NodeTypeName
nt:folder
Supertypes
nt:hierarchyNode
IsMixin
false
HasOrderableChildNodes
false
PrimaryItemName
null
ChildNodeDefinition
Name *
RequiredPrimaryType [nt:hierarchyNode]
DefaultPrimaryType null
AutoCreated false
Mandatory false
OnParentVersion VERSION
Protected false
SameNameSiblings false

A.11 nt:resource

NodeTypeName
nt:resource
Supertypes
nt:base
mix:referenceable
IsMixin
false
HasOrderableChildNodes
false
PrimaryItemName
jcr:data
PropertyDefinition
Name jcr:encoding
RequiredType STRING
ValueConstraints []
DefaultValues null

AutoCreated false
Mandatory false
OnParentVersion COPY
Protectedfalse
Multiple false
PropertyDefinition
Name jcr:mimeType
RequiredType STRING
ValueConstraints []
DefaultValue null
AutoCreated false
Mandatory true
OnParentVersion COPY
Protected false
Multiple false
PropertyDefinition
Name jcr:data
RequiredType BINARY
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory true
OnParentVersion COPY
Protected false
Multiple false
PropertyDefinition
Name jcr:lastModified
RequiredType DATE
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory true
OnParentVersion IGNORE
Protected false
Multiple false

A.12 nt:nodeType

NodeTypeName
nt:nodeType
Supertypes
nt:base
IsMixin
false

HasOrderableChildNodes
false
PrimaryItemName
null
PropertyDefinition
Name jcr:nodeTypeName
RequiredType NAME
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory true
OnParentVersion COPY
Protected false
Multiple false
PropertyDefinition
Name jcr:supertypes
RequiredType NAME
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory false
OnParentVersion COPY
Protected false
Multiple true
PropertyDefinition
Name jcr:isMixin
RequiredType BOOLEAN
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory true
OnParentVersion COPY
Protected false
Multiple false
PropertyDefinition
Name jcr:hasOrderableChildNodes
RequiredType BOOLEAN
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory true
OnParentVersion COPY
Protected false
Multiple false
PropertyDefinition

Name jcr:primaryItemName
RequiredType NAME
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory false
OnParentVersion COPY
Protected false
Multiple false
ChildNodeDefinition
Name jcr:propertyDefinition
RequiredPrimaryTypes [nt:propertyDefinition]
DefaultPrimaryType nt:propertyDefinition
AutoCreated false
Mandatory false
OnParentVersion VERSION
Protected false
SameNameSiblings true
ChildNodeDefinition
Name jcr:childNodeDefinition
RequiredPrimaryTypes [nt:childNodeDefinition]
DefaultPrimaryType nt:childNodeDefinition
AutoCreated false
Mandatory false
OnParentVersion VERSION
Protected false
SameNameSiblings true

A.13 nt:propertyDefinition

NodeTypeName
nt:propertyDefinition
Supertypes
nt:base
IsMixin
false
HasOrderableChildNodes
false
PrimaryItemName
null
PropertyDefinition
Name jcr:name
RequiredType NAME
ValueConstraints []

DefaultValues null
AutoCreated false
Mandatory false
OnParentVersion COPY
Protected false
Multiple false
PropertyDefinition
Name jcr:autoCreated
RequiredType BOOLEAN
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory true
OnParentVersion COPY
Protected false
Multiple false
PropertyDefinition
Name jcr:mandatory
RequiredType BOOLEAN
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory true
OnParentVersion COPY
Protectedfalse
Multiple false
PropertyDefinition
Name jcr:onParentVersion
RequiredType STRING
ValueConstraints [COPY, VERSION, INITIALIZE, COMPUTE, IGNORE, ABORT]
DefaultValues null
AutoCreated false
Mandatory true
OnParentVersion COPY
Protected false
Multiple false
PropertyDefinition
Name jcr:protected
RequiredType BOOLEAN
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory true
OnParentVersion COPY

Protected false
Multiple false
PropertyDefinition
Name jcr:requiredType
RequiredType STRING
ValueConstraints [STRING, BINARY, LONG, DOUBLE, BOOLEAN, DATE, NAME, PATH, REFERENCE, UNDEFINED]
DefaultValues null
AutoCreated false
Mandatory true
OnParentVersion COPY
Protected false
Multiple false
PropertyDefinition
Name jcr:valueConstraints
RequiredType STRING
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory false
OnParentVersion COPY
Protected false
Multiple true
PropertyDefinition
Name jcr:defaultValues
RequiredType UNDEFINED
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory false
OnParentVersion COPY
Protected false
Multiple true
PropertyDefinition
Name jcr:multiple
RequiredType BOOLEAN
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory true
OnParentVersion COPY
Protected false
Multiple false

A.14 nt:childNodeDefinition

NodeTypeName
nt:childNodeDefinition
Supertypes
nt:base
IsMixin
false
HasOrderableChildNodes
false
PrimaryItemName
null
PropertyDefinition
Name jcr:name
RequiredType NAME
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory false
OnParentVersion COPY
Protected false
Multiple false
PropertyDefinition
Name jcr:autoCreated
RequiredType BOOLEAN
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory true
OnParentVersion COPY
Protected false
Multiple false
PropertyDefinition
Name jcr:mandatory
RequiredType BOOLEAN
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory true
OnParentVersion COPY
Protected false
Multiple false
PropertyDefinition
Name jcr:onParentVersion
RequiredType STRING

ValueConstraints [COPY, VERSION, INITIALIZE, COMPUTE, IGNORE, ABORT]
DefaultValues null
AutoCreated false
Mandatory true
OnParentVersion COPY
Protected false
Multiple false
PropertyDefinition
Name jcr:protected
RequiredType BOOLEAN
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory true
OnParentVersion COPY
Protected false
Multiple false
PropertyDefinition
Name jcr:requiredPrimaryTypes
RequiredType NAME
ValueConstraints []
DefaultValues [nt:base]
AutoCreated false
Mandatory true
OnParentVersion COPY
Protected false
Multiple true
PropertyDefinition
Name jcr:defaultPrimaryType
RequiredType NAME
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory false
OnParentVersion COPY
Protectedfalse
Multiple false
PropertyDefinition
Name jcr:sameNameSiblings
RequiredType BOOLEAN
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory true

OnParentVersion COPY
Protected false
Multiple false

A.15 nt:versionHistory

NodeTypeName
nt:versionHistory
Supertypes
nt:base
mix:referenceable
IsMixin
false
HasOrderableChildNodes
false
PrimaryItemName
null
PropertyDefinition
Name jcr:versionableUuid
RequiredType STRING
ValueConstraints []
DefaultValues null
AutoCreated true
Mandatory true
OnParentVersion ABORT
Protected true
Multiple false
ChildNodeDefinition
Name jcr:rootVersion
RequiredPrimaryTypes [nt:version]
DefaultPrimaryType nt:version
AutoCreated true
Mandatory true
OnParentVersion ABORT
Protected true
SameNameSiblings false
ChildNodeDefinition
Name jcr:versionLabels
RequiredPrimaryTypes [nt:versionLabels]
DefaultPrimaryType nt:versionLabels
AutoCreated true
Mandatory true
OnParentVersion ABORT
Protected true

SameNameSiblings false
ChildNodeDefinition
Name *
RequiredPrimaryTypes[nt:version]
DefaultPrimaryType nt:version
AutoCreated false
Mandatory false
OnParentVersion ABORT
Protected true
SameNameSiblings false

A.16 nt:versionLabels

NodeTypeName
nt:versionLabels
Supertypes
nt:base
IsMixin
false
HasOrderableChildNodes
false
PrimaryItemName
null
PropertyDefinition
Name *
RequiredType REFERENCE
ValueConstraints [nt:version]
DefaultValues null
AutoCreated false
Mandatory false
OnParentVersion ABORT
Protected true
Multiple false

A.17 nt:version

NodeTypeName
nt:version
Supertypes
nt:base
mix:referenceable
IsMixin
false
HasOrderableChildNodes

false
PrimaryItemName
null
PropertyDefinition
Name jcr:created
RequiredType DATE
ValueConstraints []
DefaultValues null
AutoCreated true
Mandatory true
OnParentVersion ABORT
Protected true
Multiple false
PropertyDefinition
Name jcr:predecessors
RequiredType REFERENCE
ValueConstraints [nt:version]
DefaultValues null
AutoCreated false
Mandatory false
OnParentVersion ABORT
Protected true
Multiple true
PropertyDefinition
Name jcr:successors
RequiredType REFERENCE
ValueConstraints [nt:version]
DefaultValues null
AutoCreated false
Mandatory false
OnParentVersion ABORT
Protected true
Multiple true
ChildNodeDefinition
Name jcr:frozenNode
RequiredPrimaryTypes [nt:frozenNode]
DefaultPrimaryType null
AutoCreated false
Mandatory false
OnParentVersion ABORT
Protected true
SameNameSiblings false

A.18 nt:frozenNode

NodeTypeName
nt:frozenNode
Supertypes
nt:base
mix:referenceable
IsMixin
false
HasOrderableChildNodes
true
PrimaryItemName
null
PropertyDefinition
Name jcr:frozenPrimaryType
RequiredType NAME
ValueConstraints []
DefaultValues null
AutoCreated true
Mandatory true
OnParentVersion ABORT
Protected true
Multiple false
PropertyDefinition
Name jcr:frozenMixinTypes
RequiredType NAME
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory false
OnParentVersion ABORT
Protected true
Multiple true
PropertyDefinition
Name jcr:frozenUuid
RequiredType STRING
ValueConstraints []
DefaultValues null
AutoCreated true
Mandatory true
OnParentVersion ABORT
Protected true
Multiple false
PropertyDefinition
Name *

RequiredType UNDEFINED
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory false
OnParentVersion ABORT
Protected true
Multiple false
PropertyDefinition
Name *
RequiredType UNDEFINED
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory false
OnParentVersion ABORT
Protected true
Multiple true
ChildNodeDefinition
Name *
RequiredPrimaryTypes [nt:base]
DefaultPrimaryType null
AutoCreated false
Mandatory false
OnParentVersion ABORT
Protected true
SameNameSiblings true

A.19 nt:versionedChild

NodeTypeName
nt:versionedChild
Supertypes
nt:base
IsMixin
false
HasOrderableChildNodes
false
PrimaryItemName
null
PropertyDefinition
Name jcr:childVersionHistory
RequiredType REFERENCE
ValueConstraints [nt:versionHistory]

DefaultValues null
AutoCreated true
Mandatory true
OnParentVersion ABORT
Protected true
Multiple false

A.20 nt:query

NodeTypeName
nt:query
Supertypes
nt:base
IsMixin
false
HasOrderableChildNodes
false
PrimaryItemName
null
PropertyDefinition
Name jcr:statement
RequiredType STRING
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory false
OnParentVersion COPY
Protected false
Multiple false
PropertyDefinition
Name jcr:language
RequiredType STRING
ValueConstraints []
DefaultValues null
AutoCreated false
Mandatory false
OnParentVersion COPY
Protected false
Multiple false

Bibliografia

[1] Java Content Repository: Apache Jackrabbit
http://java.html.it/articoli/leggi/3167/java-content-repository-apache-jackrabbit/ (Settembre 2009)

[2] JCR or RDBMS: why, when, how?
http://www.scribd.com/doc/11163161/JCR-or-RDBMS-why-when-how

[3] JSR-000170 Content Repository for Java technology API Final Release 17 Jun, 2005
http://jcp.org/en/jsr/detail?id=170

[4] Sanmarco Informatica - Galileo ERP
http://www.sanmarcoinformatica.it/home.pag

[5] Document Management System (DMS)
http://it.wikipedia.org/wiki/Document_Management_System

[6] Alfresco
http://it.wikipedia.org/wiki/Alfresco

[7] Content Management System (CSM)
http://it.wikipedia.org/wiki/Content_management_system

[8] Jackrabbit
http://en.wikipedia.org/wiki/Apache_Jackrabbit

[9] Hippo CSM
http://en.wikipedia.org/wiki/Hippo_CMS

[10] Magnolia (CSM)
http://en.wikipedia.org/wiki/Magnolia_(CMS)

[11] Database
http://it.wikipedia.org/wiki/Database

[12] JCR: (Rapid) Content-Driven Application Development
http://www.slideshare.net/mario.cartia/jcr-20-rapid-contentdriven-application-development

[13] What is Java Content Repository
http://tim.oreilly.com/pub/a/onjava/2006/10/04/what-is-java-content-repository.html

[14] JCR API
http://jackrabbit.apache.org/jcr-api.html

[15] Specifiche JSR-000170 Content Repository for Java technology API Final Release 17 Jun, 2005
http://jcp.org/en/jsr/detail?id=170
Documentazione specifiche: jsr170-1.0.pdf.

[16] Standalone Server
http://jackrabbit.apache.org/standalone-server.html

[17] Jackrabbit Components
http://jackrabbit.apache.org/jackrabbit-components.html

[18] Jackrabbit Architecture
http://jackrabbit.apache.org/jackrabbit-architecture.html

[19] How jackrabbit works
http://jackrabbit.apache.org/how-jackrabbit-works.html

[20] Concurrency Control
http://jackrabbit.apache.org/concurrency-control.html

[21] Deployment Models
http://jackrabbit.apache.org/deployment-models.html

[22] Jackrabbit Configuration
http://jackrabbit.apache.org/jackrabbit-configuration.html

[23] Node Types
http://jackrabbit.apache.org/node-types.html

[24] Galileo
http://www-03.ibm.com/systems/it/smartmarket/Galileo-ERP.html