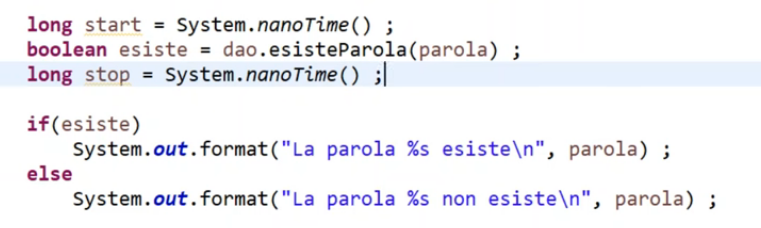
TECNICHE DI PROGRAMMAZIONE

16/04/19

Connection Pooling. Object-Relational Mapping.

La prima volta che abbiamo affrontato il tema “JDBC” avevamo visto che c’era una classe DAO che permetteva di fare query nel Database (nel nostro caso era dizionario) e c’era un metodo che ci permetteva di vedere se una parola esisteva oppure no. Adesso controlliamo quanto è efficiente questo metodo.

Isoliamo il metodo salvandolo in una variabile di tipo boolean e successivamente attraverso 2 metodi di Eclipse calcoliamo il tempo di processo e la loro differenza ci da il tempo che ha impiegato il metodo.



Notiamo che la prima volta impiega circa 1 secondo, mentre le altre volte circa 450 ms.

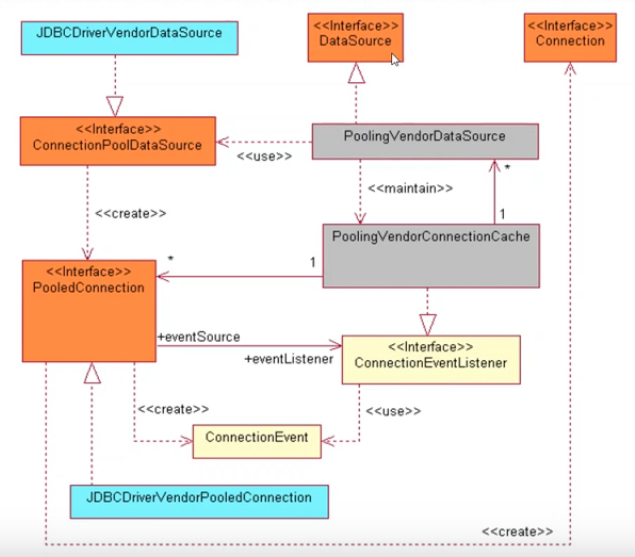
Se facciamo partire la query direttamente da Heidi, notiamo che ci mette 0,000 secondi. Il nostro main impiega 450 ms per chiamare una funzione che lancia una query la quale ci mette 0,000 secondi. Dove è finito tutto il tempo? È finito in operazioni poco utili.

Se mettessimo dei timer dopo ogni riga di codice della classe DAO, ci accorgeremmo che la maggior parte del tempo viene preso dalla connessione con il Database (getConnection).

La soluzione a questo problema è il CONNECTION POOLING.

Esso è un gruppo di connessione a disposizione di tutti che vengono aperte ad inizio programma verso il database. Quando il metodo DAO ha bisogno di fare una query, prende in prestito una delle connessioni, svolge la query e la restituisce al Pool da cui l’ha presa.

Il Pool di connessioni garantisce che ci siano sempre connessioni disponibili. A livello di classi il ConnectionPool è fatto nel seguente modo:



Il ConnectionPool pubblica un’interfaccia che si chiama DataSource (sinonimo di ConnectionPool). Noi abbiamo bisogno di una libreria che implementi il DataSource. Quest’ultimo si crea delle connessioni internamente e le da in prestito.

Quindi, da ora in avanti, se vogliamo usare questo metodo, non usiamo più la classe “DriverMenager.getConnection” per aprire una connessione, ma useremo “DataSource.getConnection”.

Quindi l’unica cosa che cambia il tutto il nostro codice è l’implementazione del getConnection (usando una libreria).

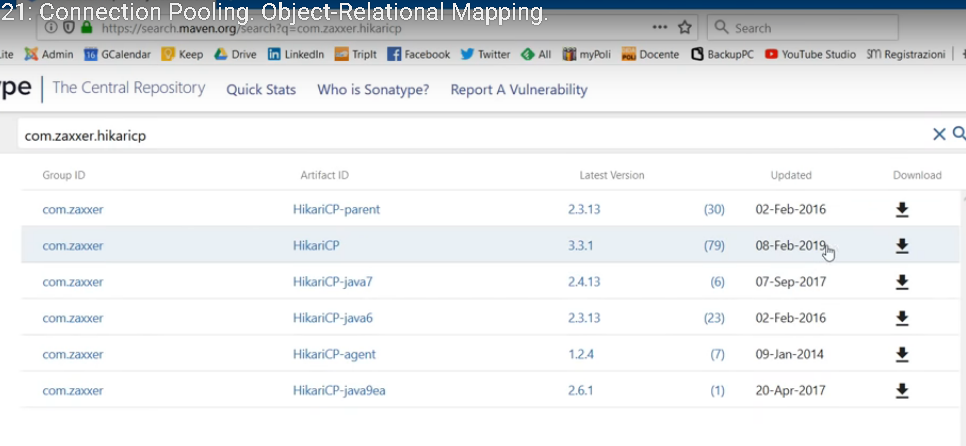
Tra le librerie più semplici da usare ce ne sono due:

* c3p0
* HikariCP (meglio)

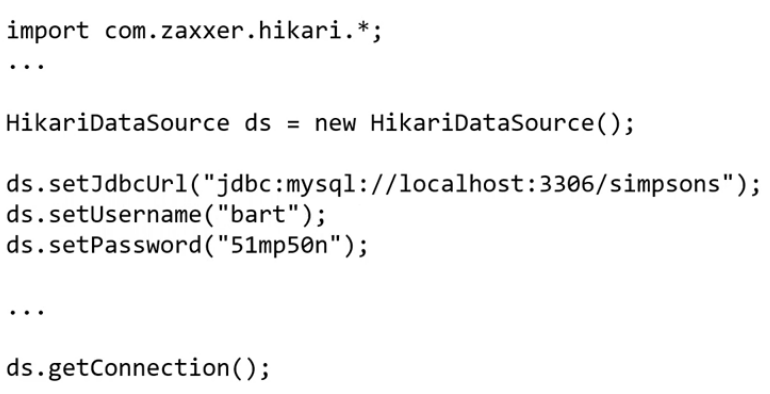
Queste librerie aggiungono la funzionalità di pooling alle connessioni JDBC verso MySQL.

Noi decidiamo quindi di usare HikariCP. Come si usa?

Come prima cosa bisogna scaricarla (vedi immagine quello con il mouse sopra).



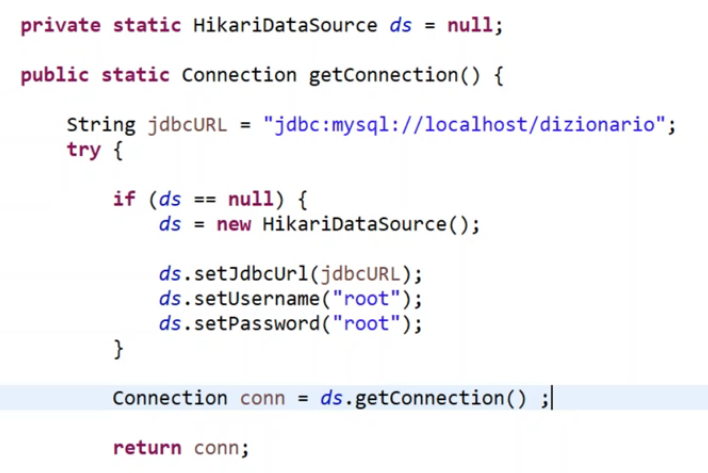
Per usare il DataSource, una volta importata la libreria, anziché usare il DriverManager, mi creo un nuovo oggetto di tipo “HikariDataSource” e a questo passo i parametri di impostazione.



Così facendo, ho inizializzato il DataSource. A questo punto, sull’oggetto DataSource, posso chiamare quante volte voglio “getConnection” che quindi non aprirà una nuova connessione ma ne prenderà una in prestito dal Pool. Ricordarsi di chiudere sempre la connessione!!

Quindi nella pratica devo prendere il mio file che ho scaricato, trasacinarlo nella cartella “lib” del programma di Eclipse e successivamente premere col tasto destro sul file e andare su “Build Path…Add to Build Patch”.

Fare attenzione, perché adesso che usiamo il DataSource, la stringa URL cambia in quanto non ci saranno più l’user e la password (vedi foto sopra per la struttura). Il programma sarà scritto nel seguente modo:



Logicamente tutte queste righe di codice fanno parte della classe DBConnect.

Se runniamo il programma ci dà un errore.



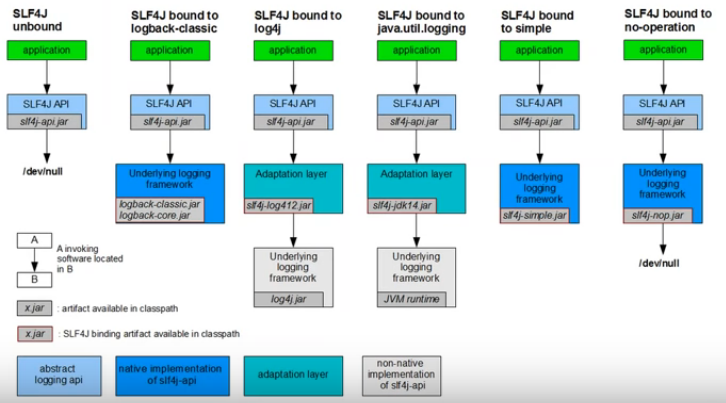
Questo errore esce fuori in quanto la libreria Hikari usa internamente una libreria di Logging. Essa rappresenta il modo serio e giusto di fare sistem.out.

Queste librerie ci permettono di decidere come trattare i messaggi di servizio del programma.

Quindi posso configurare il Logger in modo da far passare solo errori da un determinato tipo di priorità in avanti.

Noi useremo come librerie di Logging “SLF4J”. Esse si adattano ad un’unica “facade” (interfaccia). Nel nostro caso, l’interfaccia è Hikari. Quindi la libreria Hikari decide di usare una determinata “facade”, che poi si andrà ad integrare con la libreria di Logging che abbiamo deciso noi di usare.

Quest’ultime sono diverse tra loro:



La libreria di Logging semplice (SLF4J bound to simple) stampa sulla console.

Noi abbiamo bisogno di due “jar”. Il primo è slf4j che è la “facade”. Dietro alla facade possiamo avere o la libreria di Logging semplice (5°colonna), oppure, se non hai altre librerie (1°colonna), i messaggi vanno persi. Nel caso in cui io abbia già una mia libreria (log4j 3° colonna), mi viene chiesto di mettere un piccolo adapter, cioè un qualche cosa che trasla le chiamate definite a livello di facade nelle chiamate effettive implementate nella libreria sottostante.

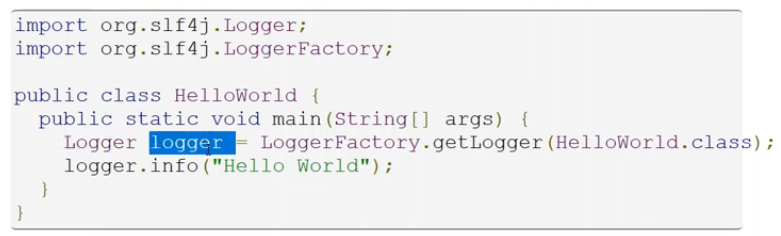


Noi andiamo in download sul sito ufficiale e scarichiamo lo zip (mouse sopra).

All’interno dello zip che scarichiamo, ci sono una serie di file Jar. “slf4j-api-1.7.26.jar”, lo conosciamo ed è la facade e dobbiamo sicuramente aggiungerla al nostro progetto. Oltre a questa dobbiamo implementare una sola delle altre librerie che di fatto si adatta al sistema di Logging che vogliamo usare. Se vogliamo loggare sulla console prenderemo “slf4j-simple-1.7.26.jar”.

Noi quindi decidiamo di copiare la “api” e la “simple” nella cartella lib del nostro progetto e successivamente le aggiungiamo al Build Path.

L’utilizzo della classe logger all’interno di un programma è molto semplice e segue questa struttura:



Quindi creiamo un oggetto logger relativo ai messaggi che la classe “HelloWorld” dovrà emettere. Per stampare i messaggi a video quindi adesso non faremo piu “System.out” ma “logger.info” se vogliamo dare un’informazione, “logger.warn” se vogliamo dare un warning, “logger.err” se vogliamo indicare un errore e così via…

Se voglio stampare dei messaggi che però ci sono solo in fase di debug (dopo non voglio più vederli), allora scriverò “logger.debug”.

Il secondo argomento è quello dell’Object-Relational Mapping.

Noi, ora come ora, abbiamo 2 mondi distinti:

* Il mondo di Java basato sugli oggetti.
* Il mondo del Database basato sulle tabelle relazionali.

Quindi noi dobbiamo prendere dei dati che sono organizzati in modo tabulare e usarli all’interno di un programma che lavora ad oggetti. Per far questo noi ci diamo delle regole di corrispondenza.

L’obiettivo di queste librerie, chiamate librerie ORM, è quello di prendere la struttura delle tabelle del database e creare delle classi Java che contengono i nomi e i campi giusti.

Ci sono delle regole da rispettare quando usiamo delle librerie ORM:

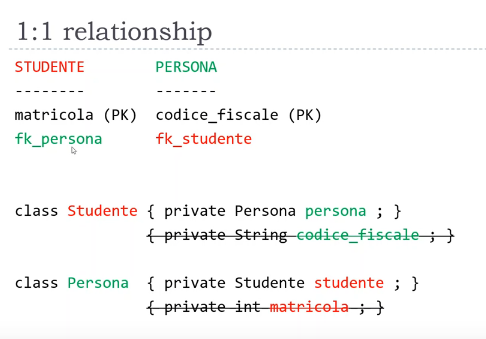
1. Creare una classe Java Bean per ogni entità (tabella) del database.
2. Le tabelle del database che servono a rappresentare delle relazioni MOLTI a MOLTI non devono avere la loro classe Java Bean.
3. Il nome della classe Java Bean deve corrispondere con il nome della tabella.
4. La classe deve avere una proprietà per ogni colonna della tabella e con lo stesso nome. Deve inoltre corrispondere il tipo di dato (con la giusta conversione). Questo ragionamento non va fatto con le chiavi esterne.

Ogni classe che creiamo avrà un costruttore che riceve tutti i campi della tabella (tranne chiavi esterne). Aggiungiamo inoltre per ogni proprietà i Get( ) e Set( ).

Come ultima cosa definiamo l’equals e l’hashCode usando esattamente gli attributi che definiscono la Chiave Primaria della tabella.

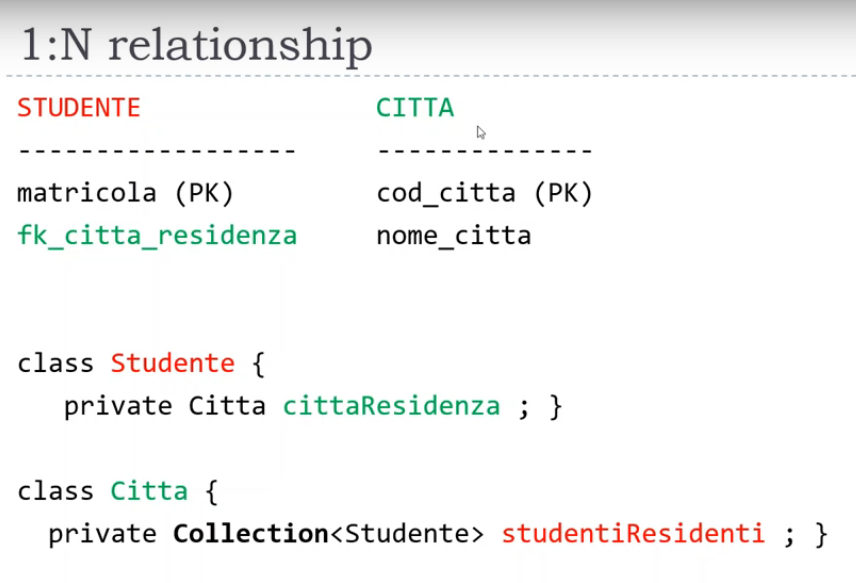
Bisogna fare attenzione perché su Java non dobbiamo rappresentare tutte le relazioni, ma solamente quelle che ci servono.

Se ho una relazione con cardinalità 1, allora metterò un attributo nella mia classe che contiene un riferimento all’altra classe.



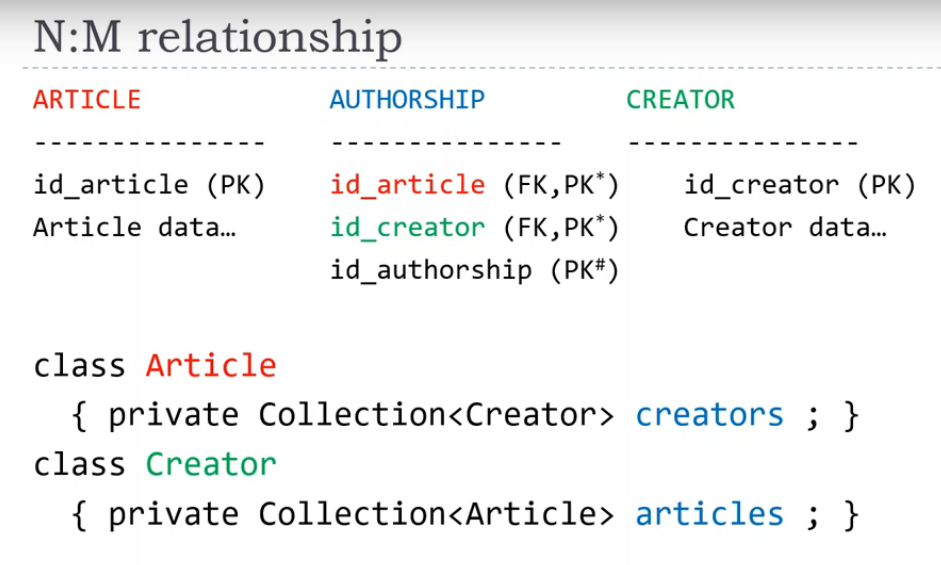
Nella nostra classe quindi avremo una variabile matricola (non rappresentata nell’immagine) e una variabile fk\_persona che è il riferimento all’altra tabella.

Nelle relazioni 1 a molti, invece, avrò una collection che dovrà essere riempita con i riferimenti agli oggetti corrispondenti (più faticoso). Queste collection dovrebbero essere dei Set o una List.



Da studente a Città abbiamo 1:1 quindi come prima. Da città a Studente invece 1 a molti e quindi usiamo una collection.

Il caso più complicato è quello della relazione molti:molti. In SQL ci aspettiamo di avere una tabella in più.



La relazione AUTHORSHIP collega le due tabelle con relazione molti:molti e presenta una chiave primaria composta con le due chiavi esterne delle altre due tabelle ( ARTICLE e CREATOR) oppure ne creiamo una nuova (id\_authorship).

In questo caso avremo comunque due classi (NON c’è la classe per la tabella che collega le due classi) e entrambe avranno una collection.

FINE.