package com.twitter.servo.database

import com.twitter.servo.repository.\_

import com.twitter.util.Future

import scala.collection.mutable.{HashMap, HashSet, ListBuffer}

import scala.collection.generic.Growable

object Database {

/\*\*

\* Construct a KeyValueRepository wrapping access to a database.

\*

\* Data retrieved as a row from the query is passed to a Builder producing a

\* (Key, Row) tuple. Once all rows have been processed this way it is passed as a

\* sequence to a post-query function that can perform actions (aggregation usually)

\* and produce a final sequence of (Key, Value).

\*

\* @tparam Q

\* how we'll be querying the this repository

\*

\* @tparam K

\* the key used for looking data up

\*

\* @tparam R

\* each entry from the the database will be represented as an instance of R

\*

\* @tparam V

\* the repository will return a V produced by processing one or more Rs

\*

\* @param database

\* A database used to back the KeyValueRepository being built.

\*

\* @param dbQuery

\* A database query for fetching records to be parsed into objects of type

\* Row. The query string can contain instances of the character '?' as

\* placeholders for parameter passed into the `Database.select` calls.

\*

\* @param builder

\* A Builder that builds (K, Row) pairs from ResultSets from the database

\*

\* @param postProcess

\* A function which can manipulate the Seq[(K, Row)] that is returned from the

\* database. Useful for aggregating multi-mapped K, V pairs where V holds a

\* container with multiple values for the same key in the database. This function

\* should not manipulate the list of keys; doing so will result in Return.None

\* elements in the ensuing KeyValueResult.

\*

\* AggregateByKey has a basic implementation that groups R objects by a

\* specified identifier and may be useful as a common impl.

\*

\* @param selectParams

\* A function that is applied to the distinct keys in a repository query.

\* The result is passed to `Database.select` to be used for filling in

\* bind variables in dbQuery. By default, the repository query is passed

\* directly to the select. The use cases for this function are situations

\* where the SELECT statement takes multiple parameters.

\*

\* Example:

\* // A repository that takes Seq[Long]s of userids and returns

\* // Item objects of a parameterized item type.

\* Database.keyValueRepository[Seq[Long], Long, Item, Item](

\* database,

\* "SELECT \* FROM items WHERE user\_id IN (?) AND item\_type = ?;",

\* ItemBuilder,

\* selectParams = Seq(\_: Seq[Long], itemType)

\* )

\*/

def keyValueRepository[Q <: Seq[K], K, R, V](

database: Database,

dbQuery: String,

builder: Builder[(K, R)],

postProcess: Seq[(K, R)] => Seq[(K, V)] =

(identity[Seq[(K, V)]] \_): (Seq[(K, V)] => Seq[(K, V)]),

selectParams: Seq[K] => Seq[Any] = (Seq(\_: Seq[K])): (Seq[K] => collection.Seq[Seq[K]])

): KeyValueRepository[Q, K, V] =

query => {

if (query.isEmpty) {

KeyValueResult.emptyFuture

} else {

val uniqueKeys = query.distinct

KeyValueResult.fromPairs(uniqueKeys) {

database.select(dbQuery, builder, selectParams(uniqueKeys): \_\*) map postProcess

}

}

}

}

/\*\*

\* A thin trait for async interaction with a database.

\*/

trait Database {

def select[A](query: String, builder: Builder[A], params: Any\*): Future[Seq[A]]

def selectOne[A](query: String, builder: Builder[A], params: Any\*): Future[Option[A]]

def execute(query: String, params: Any\*): Future[Int]

def insert(query: String, params: Any\*): Future[Long]

def release(): Unit

}

object NullDatabase extends Database {

override def select[Unit](query: String, builder: Builder[Unit], params: Any\*) =

Future.value(Seq.empty[Unit])

override def selectOne[Unit](query: String, builder: Builder[Unit], params: Any\*) =

Future.value(None)

override def release() = ()

override def execute(query: String, params: Any\*) =

Future.value(0)

override def insert(query: String, params: Any\*) =

Future.value(0)

}

object AggregateByKey {

def apply[K, R, A](

extractKey: R => K,

reduce: Seq[R] => A,

pruneDuplicates: Boolean = false

) = new AggregateByKey(extractKey, reduce, pruneDuplicates)

/\*\*

\* In the event that the item type (V) does not carry an aggregation key then we can have

\* the Builder return a tuple with some id attached. If that is done then each Row from the

\* builder will look something like (SomeGroupId, SomeRowObject). Because we tend to minimize

\* data duplication this seems to be a pretty common pattern and can be seen in

\* SavedSearchesRepository, FacebookConnectionsRepository, and UserToRoleRepository.

\*

\* @tparam K

\* The type for the key

\* @tparam V

\* The type of a single element of the list

\* @tparam A

\* The object we'll aggregate list items into

\* @param reduce

\* A function that combines a seq of V into A

\* @param pruneDuplicates

\* If set this ensures that, at most, one instance of any given V will be passed into reduce.

\*/

def withKeyValuePairs[K, V, A](

reduce: Seq[V] => A,

pruneDuplicates: Boolean

): AggregateByKey[K, (K, V), A] =

new AggregateByKey(

{ case (k, \_) => k },

values => reduce(values map { case (\_, v) => v }),

pruneDuplicates

)

}

/\*\*

\* Basic aggregator that extracts keys from a Row, groups into a Seq by those keys, and

\* performs some reduction step to mash those into an aggregated object. Order is not

\* necessarily kept between the retrieving rows from the database and passing them into

\* reduce.

\*

\* @tparam K

\* the type used by the item on which we aggregate rows

\*

\* @tparam R

\* object that a single row of the query will be represented as

\*

\* @tparam A

\* what we collect groups of R into

\*

\* @param extractKey

\* function to extract a key from a row object

\*

\* @param reduce

\* function that can take a sequence of rows and combine them into an aggregate

\*

\* @param pruneDuplicates

\* if set this will ensure that at most one copy of each R will be passed into reduce (as

\* determined by R's equal method) but will pass the input through a set which will

\* likely lose ordering.

\*/

class AggregateByKey[K, R, A](

extractKey: R => K,

reduce: Seq[R] => A,

pruneDuplicates: Boolean = false)

extends (Seq[R] => Seq[(K, A)]) {

override def apply(input: Seq[R]): Seq[(K, A)] = {

val collectionMap = new HashMap[K, Growable[R] with Iterable[R]]

def emptyCollection: Growable[R] with Iterable[R] =

if (pruneDuplicates) {

new HashSet[R]

} else {

new ListBuffer[R]

}

input foreach { element =>

(collectionMap.getOrElseUpdate(extractKey(element), emptyCollection)) += element

}

collectionMap map {

case (key, items) =>

key -> reduce(items toSeq)

} toSeq

}

}