



Originally Published on TheServerSide.com

June 12, 2002

Revison Published February 27, 2003

Introduction

Internet search applications are now ubiquitous on the web. In a search application a user enters some form of search criteria and the application returns search hits that match the criteria. Depending upon the nature of the application the number of hits could range from very small to practically infinite. Search engines for Internet content or large eCommerce product catalogs are examples of applications that could return extremely large numbers of hits. In this article, we describe a J2EE design pattern for efficiently processing Internet searches and quickly displaying the results to the user.

Problem

A typical Internet search application will accept user search criteria and use it to retrieve matching items from a data store. In applications such as a web search engine the number of matching hits could be unbounded. The hits are displayed one page at a time and a PREVIOUS and NEXT button is presented to allow a user to page forward and backward through the results. There may be many users performing searches at the same time.

The first page of results must be returned quickly. Paging forward and backward must also be fast. Because of the large number of concurrent users, the overhead for performing a search and displaying a page of results must be low.

A number of common strategies exist for processing large search result sets. These include:

- **EJB Find Method:** This approach involves using entity EJBs to represent the objects returned by the search. An EJB find method would be defined on the EJB to perform the search. The EJB find method would then return the list of EJB instances satisfying the search criteria. Although supported by the EJB specification, this approach is not practical because of the high cost of instantiating EJB instances and the high cost of communicating with EJBs. Entity EJBs are generally not suited for modeling fine-grained objects such as the objects returned by an Internet search. They are better suited for modeling coarse-grained objects. The EJB V2.0 specification introduced the notion of local interfaces to EJBs to provide lightweight access from co-located clients of the EJBs. The overhead of instantiating and accessing local interface EJBs is presumably low enough to make them better suited for modeling fine grained objects. However, the find method would still incur a high cost to instantiate the entire set of EJBs that satisfy the search criteria.

- **Fetch all hits into a collection object before displaying first page:** In this approach the entire search result set is retrieved and placed in a large collection object. Requests from the client for a subset of results are satisfied by copying a subset of the objects in the main collection into a new smaller collection that is returned to the client. For large result sets the cost and time it takes to retrieve all the results into a collection before being able to satisfy the first request for a subset of the results is prohibitively high. This is the strategy used in the Value List Handler design pattern [CJP]. A variation of this strategy is to fetch pointers to the result set items and to place them in a large collection object. The pointers are used to retrieve the items as each page is displayed. The cost of retrieving pointers can still be very high for large result sets.

- **Re-execute the search query every time the client requests a subset of the hits:** In this approach a request for a subset of the hits by the client is satisfied by re-executing the entire search and fetching only as many items as are necessary to satisfy the user request. Most search applications return the hits in a specified order. This means that requests for hits that occur latter in the return order will require retrieving and skipping over hits that appear earlier in the return order. The overhead of re-executing the query along with the need to fetch and step over a larger and larger number of hits with each client request for data makes this approach impractical for expensive search queries that return large result sets.

- **Fetch several pages of hits and store them:** This approach involves executing the query and storing a subset of the hits in a collection object. Typically, enough hits to satisfy requests to display several pages worth of results would be stored. A request for a subset of the hits by a client would be satisfied by first checking if the hits are in the saved collection object. If so they are retrieved from there. If not, the query is re-executed and the contents of the original collection are replaced with a new set of hits. The new set would include the hits currently being requested by the client. This strategy can be efficient if users tend to browse through only the first few pages worth of hits. Of course, as soon as the user moves to a page that requires hits that are not in the cached collection, there will be a delay due to the re-execution of the query and fetching of the hits.

Solution

Use a generalized interface to abstract away the underlying implementation details of the search and list processing logic. Provide an implementation of this interface to support efficient retrieval of large result sets from relational databases.

A generalized list handling interface is used to encapsulate the underlying search and retrieval implementation details. The interface is general enough to support virtually any type of underlying implementation and data store. To address the need for retrieving search results from a relational database using JDBC there is a specialization of the generalized list handling interface which is used for relational database access. To address the requirements for speed and efficiency an efficient implementation of the specialized relational database interface is provided.

Structure

Figure 1 shows the class diagram for the Data List Handler pattern.

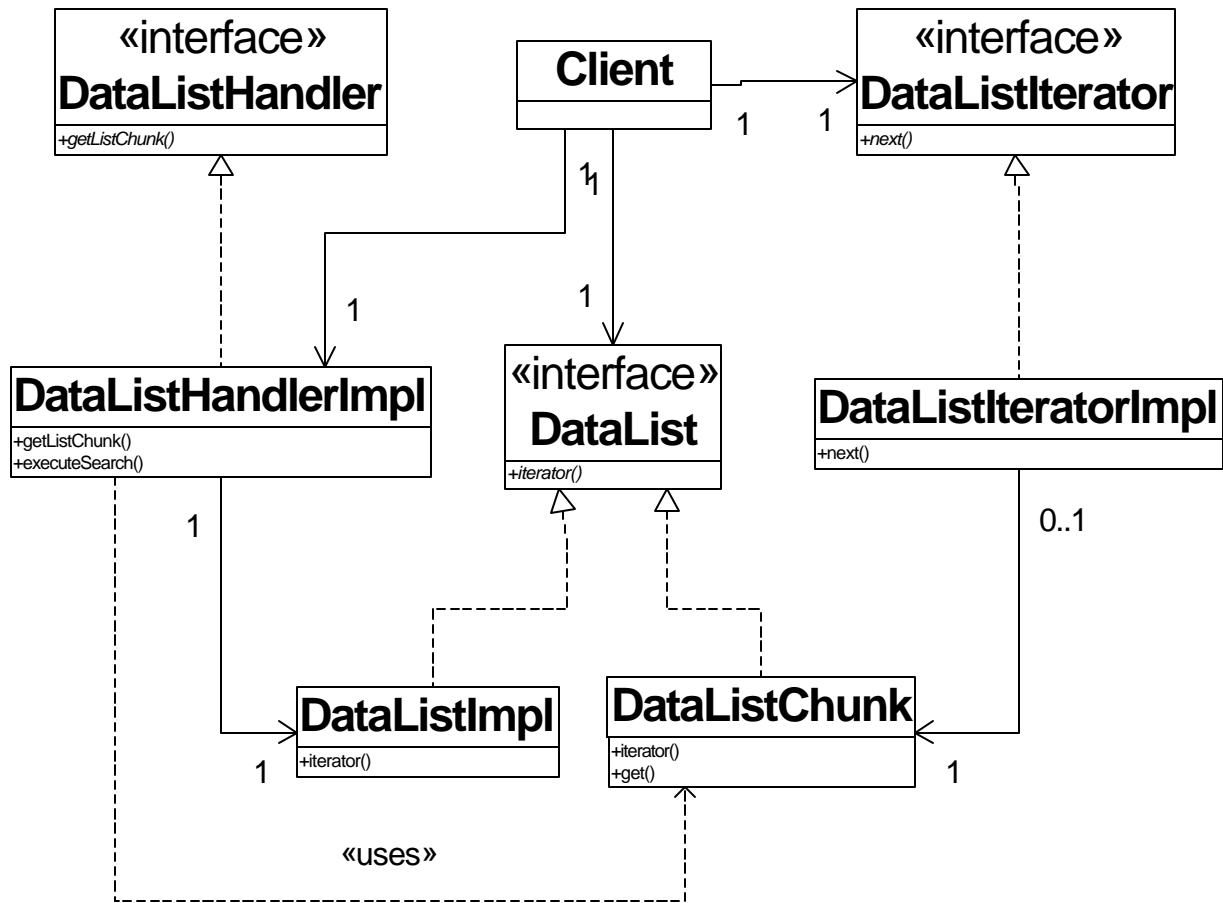


Figure 1. Data List Handler Class Diagram

Participants and Collaborations

Figure 2 shows the sequence diagram for the interaction of a client with a Data List Handler.

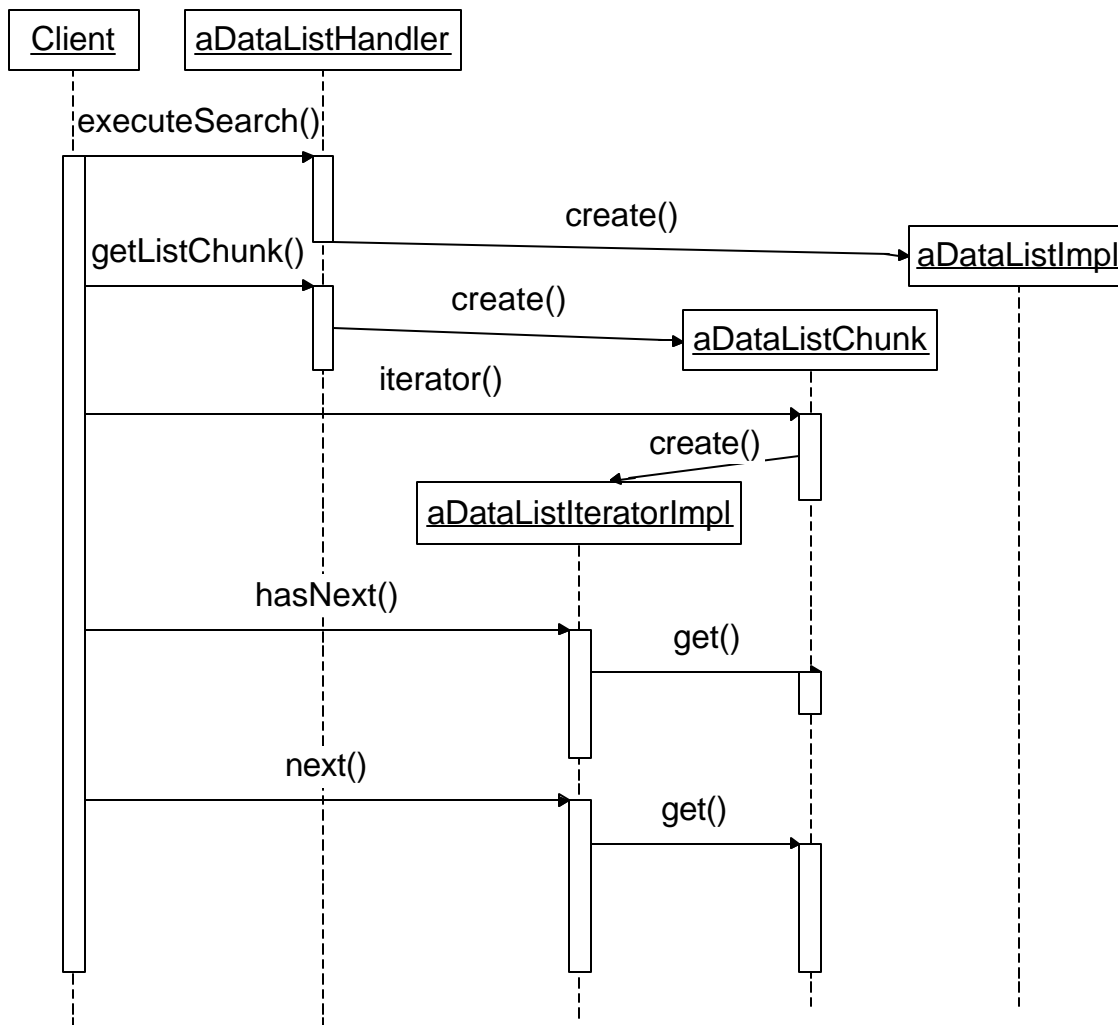


Figure 2. Data List Handler Sequence Diagram

Client

This represents a client of the Data List Handler. The client asks the Data List Handler to execute a search and then repeatedly requests a subset of the search results. The client can either be a Servlet tier component that handles presentation of the results or it can be an EJB that might then channel the results back to another client in the Servlet tier.

DataList

This is a general list interface that is used to represent a collection of search results. It is designed to be more flexible than the standard Java List interface. The greater flexibility allows for underlying implementations that could perform more efficiently than would otherwise be possible. The difference between the List and DataList interfaces is most evident when comparing the signatures of the get methods of the two interfaces. It includes the following example methods:

- DataListIterator iterator() returns an iterator object that allows a client to step through the items in the list sequentially.

- void close() signals that the client no longer needs the results and that any resources being held can be released.
- Object get(int index, Object item) returns the item at a particular position in the collection. The item argument is an empty instance of a Class that represents an item in the collection. The method is designed to populate the state of the empty instance with values that reflect the state of the item requested. It then returns the populated item instance. The advantage of the get method is that it allows for underlying implementations in which the items in the collection may not exist physically and are materialized at the time they are requested by a client. By passing in an empty object in which to place the requested item we avoid the overhead of having to instantiate a new object with each get request. Without passing in an empty object, it would be difficult to avoid the overhead of instantiating a new object.
- boolean isEmpty() determines if the list is empty

DataListIterator

An interface analogous to the standard Java Iterator interface. The methods include:

- boolean hasNext() is strictly analogous to the List.hasNext method
- Object next(Object obj) obtains the next element in the underlying DataList. Like the DataList.get method, it accepts an instance of a Class that represents items in the collection and populates it with state that reflects the item being requested. It then returns the same object instance. This method differs from the standard Iterator.next method for the same reasons outlined above for the difference between the DataList.get method and the List.get method.

DataListHandler

This is a generalized list handling interface that is used by a client application to retrieve a subset of the results from an internet search. Typically the client would retrieve enough hits to display on a single page. It includes the following example methods:

- DataList getListChunk(int startIndex, int count) returns a subset of the search result hits.
- boolean elementExists(int index) determines if an item at a particular position in the list exists.
- void close() signals that the client no longer needs the results and that any resources being held can be released.

DataListImpl

This class implements the DataList interface to represent the entire collection of hits that satisfy a user search request.

DataListChunk

This class implements the DataList interface to represent a subset of the hits stored in a DataListImpl instance. An instance of this class would be created to satisfy a request from a client for a subset of the hits in the collection to present a page worth of hits to the user. A DataListChunk would be returned by the DataListHandler.getListChunk method.

DataListIteratorImpl

Implements the DataListIterator interface. A client uses it to access and traverse the elements of a DataListImpl or a DataListChunk. It is created by the iterator method of a DataListImpl or DataListChunk. Typically a client would obtain a DataListChunk instance from a DataListHandlerImpl and then call the iterator method of the DataListChunk to get a

DataListIteratorImpl instance that it can then use to traverse the elements of the DataListChunk.

DataListHandlerImpl

This class implements DataListHandler. The client interacts directly with this class to request a search. The DataListHandler uses a Data Access Object to perform a data store search and to create a DataListImpl instance to store the search results. It responds to getListChunk method requests by creating a DataListChunk instance and returning it to the client.

Relational Database Access Strategy

The strategy described here will focus on an efficient implementation of the Data List Handler pattern for accessing large search result sets from a relational database. This strategy involves the use of another layer of interfaces below the top level Data List pattern interfaces. Whereas the outer level Data List interfaces are designed for processing search result sets returned from any data source, the interfaces described here are tailored for processing result sets returned from relational databases. The classes that participate in this strategy implement the lower level relational database specific interfaces rather than the outer level Data List interfaces. The structure of this strategy is given in figure 3 and the sequence diagram is given in figure 4.

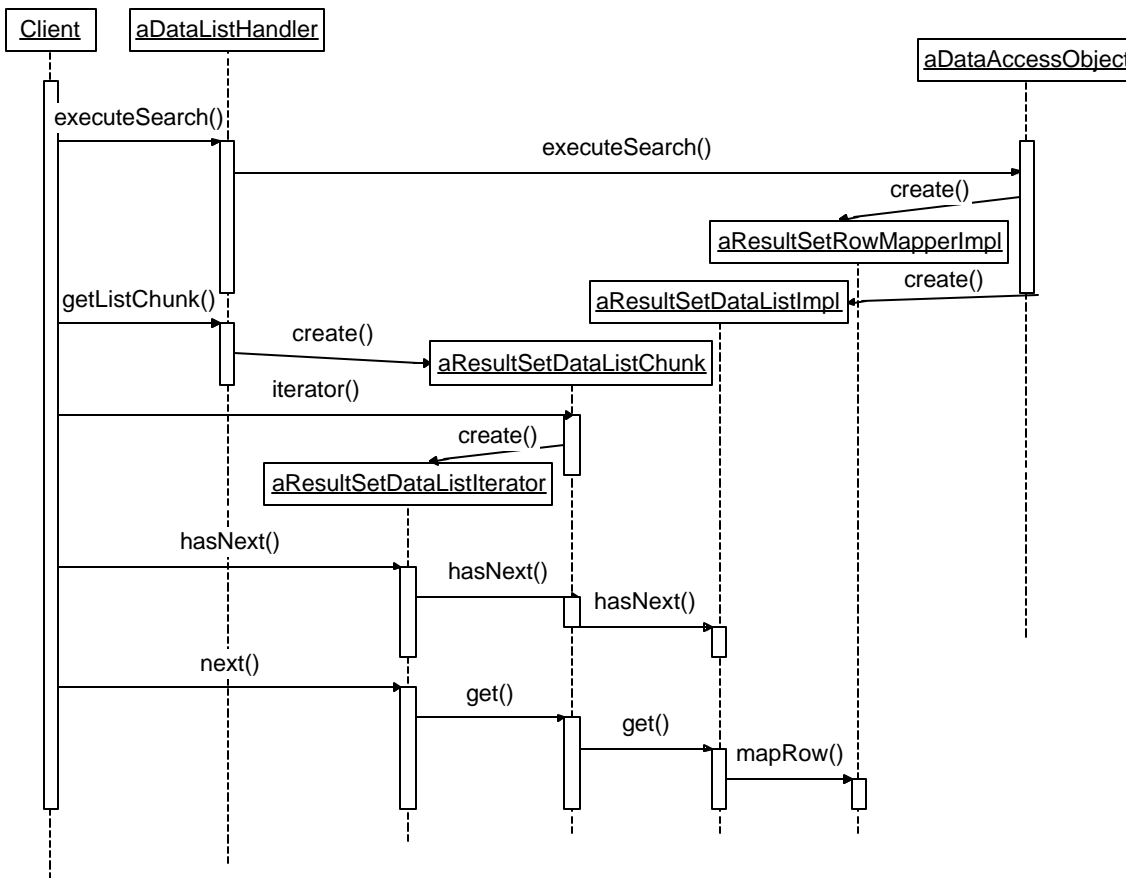


Figure 4. Relational Database Access Strategy Sequence Diagram

A common characteristic of strategies that do not perform efficiently is that they open and close JDBC Connection and ResultSet objects with each request for search hits. The strategy described here, on the other hand, keeps the underlying Connection and ResultSet objects open across requests. By keeping the ResultSet open across requests, it is possible to only retrieve as many rows from the database as necessary to satisfy a given call to `getListChunk`. We don't have to incur the overhead of fetching items that the client is not requesting. In order to support random access to any subset of items in the search result set, the scrollable ResultSet feature of JDBC 2.0 is used.

Before presenting the strategy participants and collaborations some background on JDBC 2.0 scrollable ResultSet's will be given.

JDBC 2.0 Scrollable ResultSet's

JDBC 2.0 introduced the notion of a ResultSet type. They types are:

- **TYPE_FORWARD_ONLY** (Default, JDBC 1.0 behavior) The ResultSet is not scrollable. It's cursor moves forward only from before the first row to the last.
- **TYPE_SCROLL_INSENSITIVE** The ResultSet is scrollable. Its cursor can move backwards, forwards, and be positioned either to an absolute row number or to a relative row number.

- **TYPE_SCROLL_SENSITIVE** The ResultSet is scrollable like TYPE_SCROLL_INSENSITIVE. Unlike TYPE_SCROLL_INSENSITIVE, changes committed by other users while the ResultSet is open are visible.

The following methods are available for scrollable ResultSets.

- public void previous() moves to the previous row
- public boolean absolute(int index) positions to an absolute row number
- public void beforeFirst() moves to before the first row in the ResultSet
- public void afterLast() positions to after last row in ResultSet
- public void first() positions to first row
- public void last() positions to last row
- public int getRow() returns current row number. Rows are numbered starting with 1.
- public boolean isBeforeFirst() indicates if cursor is positioned before first row
- public boolean isAfterLast() indicates if cursor is positioned after last row
- public boolean isFirst() indicates if current row is first row
- public boolean isLast() indicates if current row is last row

The efficiency of a ResultSet implementation depends on the vendor. For example, Oracle has a very efficient implementation. It implements scrollable ResultSets in the JDBC driver layer by fetching rows from the database in a forward direction only. As rows are fetched they are stored in a client-side memory cache maintained by the JDBC driver. Requests for rows that have already been retrieved from the database and placed in the client side cache are satisfied by accessing them from the local cache. The driver only goes to the database to satisfy a request for a row that isn't already in the cache.

Strategy Participants and Collaborations

ResultSetDataList

This is an interface that extends the DataList interface. This interface implies the concept of a cursor and a corresponding current row for the underlying collection. Even though collections don't normally have such a notion, it is important to expose the notion of a current row in the interface in order to accommodate underlying efficient implementations. The interface has the following example methods that are tailored for relational database access using JDBC:

- boolean hasNext() determines if there exists a row after the current row.
- void beforeFirst() determines if the collection cursor is positioned before the first row of the list.
- boolean absolute(int index) positions the cursor to a specific row number. Rows are numbered starting from 0.
- boolean elementExists(int index) indicates if the item at position index exists

ResultSetDataListImpl

This class implements `ResultSetDataList` as a virtual collection that represents the entire result set of the client query. It contains a scrollable JDBC `ResultSet` object that can also be thought of as a virtual collection. An implementation of a scrollable JDBC `ResultSet` like that of Oracle's defers fetching of rows from the database until the cursor is positioned to the requested row. This reduces the time it takes to return the first set of rows in the result set to the client to an absolute minimum. It also avoids the overhead associated with implementations that retrieve all rows up front.

In addition to containing a `ResultSet` object, this class also contains a reference to the JDBC `Connection` object used to create the `ResultSet`. This is necessary to preserve the `ResultSet` across client requests and is a key element of the strategy.

This class implements the `get` method of the `DataList` interface by positioning the cursor of the underlying `ResultSet` object to the requested row number and using `ResultSet.get` methods to fetch the field values of the row and placing them in instance variables of the input `Object` argument. This illustrates the point made earlier about why the `DataList.get` method differs from the standard `List.get` method. Because there is no physical underlying collection of objects, a `List` interface style `get` method would require that a new object be instantiated to satisfy each `get` request.

ResultSetDataListChunk

This class implements the `ResultSetDataList` interface as a virtual collection that represents a subset of the elements in another `ResultSetDataList`. The subset it represents is the set of rows requested by a client with a call to `DataListHandler.getListChunk`. Because a `ResultSetDataListChunk` is itself a `ResultSetDataList` it is possible to create a `ResultSetDataListChunk` that represents a subset of another `ResultSetDataListChunk`. Because it is a virtual collection with no physical underlying collection there is little instantiation overhead.

ResultSetIterator

This class is an implementation of `DataListIterator` that is tailored to iterate over `ResultSetDataList`s. This class provides the key to understanding why the `ResultSetDataList` interface exists at all. If there were no `ResultSetDataList` interface, `ResultSetDataListImpl` and `ResultSetDataListChunk` would implement the `DataList` interface directly. Then it would be necessary to have two separate implementations of the `DataListIterator` interface, one to iterate over a `ResultSetDataListImpl` and another to iterate over a `ResultSetDataListChunk`. By introducing the `ResultSetDataList` interface below the `DataList` interface which `ResultSetDataListChunk` and `ResultSetDataListImpl` both implement, we can get by with just one iterator class. The single `ResultSetIterator` class iterates over the interface rather than over any specific implementation of the interface.

The implementation of the `ResultSetIterator` and the `ResultSetDataListImpl` class shown in the sample code sections latter in the document assumes that there will only ever be one instance of a `ResultSetIterator` for a given `ResultSetDataList` instance. This is clearly not the normal behavior of a standard Java `Iterator` object. In fact, one of the main motivations for the existence of an `Iterator` object is to allow concurrent traversals of the same list. The reason for imposing this constraint on the `ResultSetIterator` implementation is that this allows us to assume that the cursor of the underlying JDBC `ResultSet` is already in the correct position when executing the `ResultSetIterator.next` method. Consequently, the code is simpler and the performance is slightly better. Among the changes required to support multiple concurrent `ResultSetIterator` objects for the same `ResultSetDataList` instance is to make some of the methods of the `ResultSetDataListImpl` and `ResultSetDataListChunk` classes synchronized. For example, the `get` method would have to be synchronized.

ResultSetRowMapper

The `ResultSetDataListImpl.get` method accepts an `Object` as an input argument. The `ResultSetDataListImpl` is a generic

class with no specific knowledge of the structure of the rows returned by the underlying JDBC result set. Thus, it needs some way to map the columns of a row of data to the instance variables of an Object passed in as an argument to the get method. This is where the `ResultSetRowMapper` interface comes in. It maps columns of a row of data to instance variables of an Object that represents an item in a search result set. It has the following method:

`Object mapRow(ResultSet rs, Object item)` retrieves the columns of a row of data using `ResultSet.getter` methods and stores them in instance variables of `item`. It returns the populated item object.

ResultSetRowMapperImpl

This class is an implementation of `ResultSetRowMapper` for a specific type of object. There is an implementation of this class for each type of object stored in a search result set. All items in a search result set must be of the same type.

DataListHandlerImpl

The `DataListHandlerImpl` class which implements the `DataListHandler` interface uses a Data Access Object to perform a database search and to create a `ResultSetDataListImpl` instance to store the search results.

DataAccessObject

This is an application specific class that executes the query against the database and creates a `ResultSetDataListImpl` object to store the JDBC `ResultSet`. It passes an application specific `ResultSetRowMapperImpl` instance to the constructor of `ResultSetDataListImpl` needed to map `ResultSet` rows to an object that represents an item in the search result.

Connection Expiration Strategy

A key element of the relational database access strategy is the preservation of database connections in the `ResultSetDataListImpl` object across client requests. This is problematic because database connections may remain open for extended periods of time and there is overhead in maintaining open database connections. We address the problem by extending the relational database access strategy to ensure that database connections are short lived. The structure of this strategy is given in figure 5 and the sequence diagram is given in figure 6. Some of the classes from the relational database access strategy have been omitted to keep the diagrams readable. The steps after the `getListChunk` call sequence have been omitted from figure 6 because they are the same as those in figure 4.

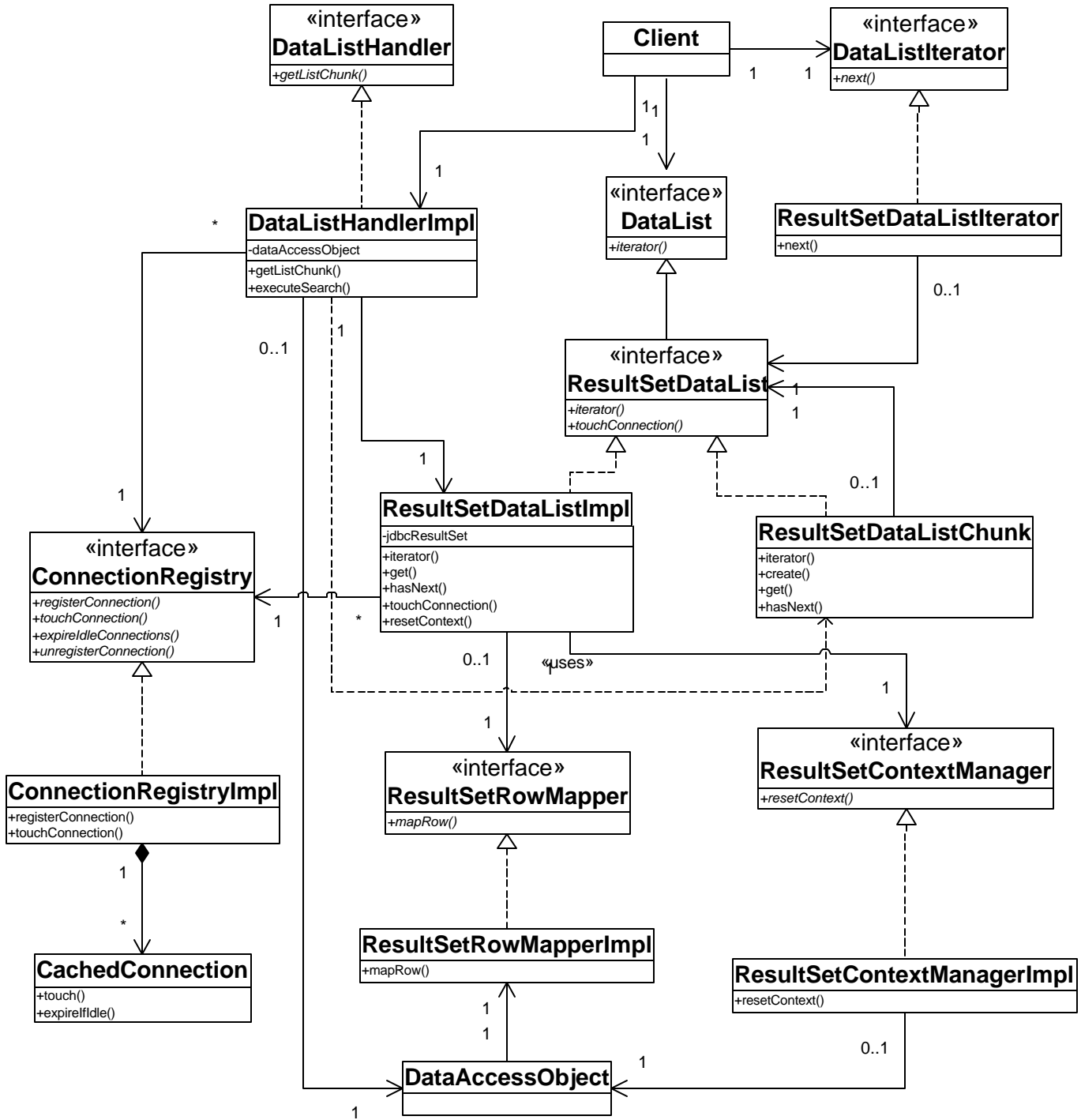


Figure 5. Connection Expiration Strategy Class Diagram

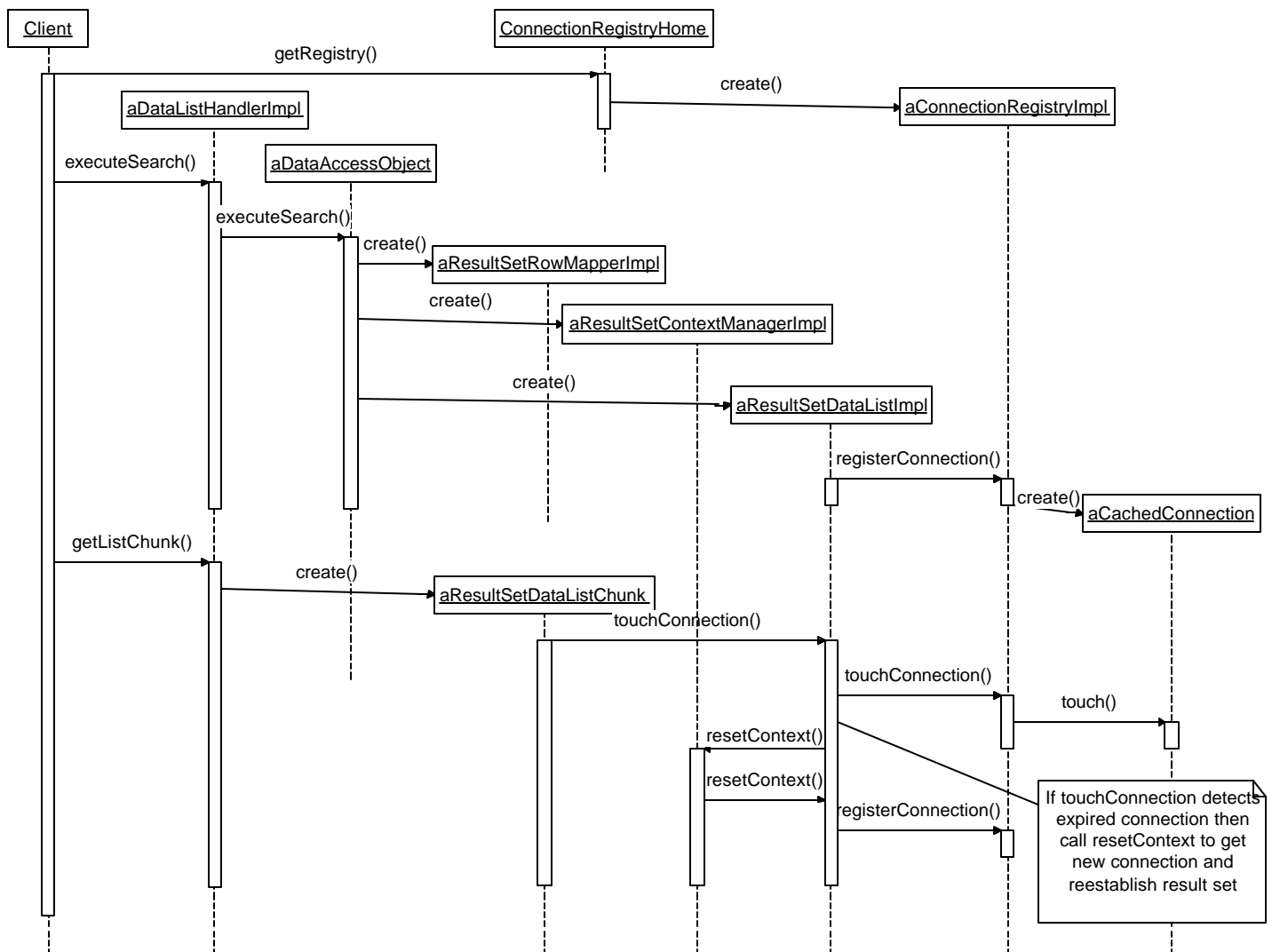


Figure 6. Connection Expiration Strategy Sequence Diagram

Strategy Participants and Collaborations

We will describe new or modified participants and collaborations. See the Relational Database Access Strategy for a description of the unchanged participants of that strategy.

ConnectionRegistry

This is an interface that represents a repository of open database connections. It has the following example methods:

- void registerConnection(Connection conn) registers a connection in the repository.
- void unRegisterConnection(Connection conn) removes a connection from the repository.

- void touchConnection(Connection conn) signals that an operation has been performed on a connection in the repository.
- void expireIdleConnections() closes idle connections and removes them from the repository.

CachedConnection

This class wraps a database Connection in the registry to provide a timestamp that represents the last time an operation was performed on the Connection. It has the following example methods:

- void touch() updates the timestamp to indicate that an operation was performed on the contained Connection
- void expireIfIdle(long maxIdleMillisecs) closes the contained Connection object and flags the CachedConnection object as expired if the timestamp indicates that the Connection object has been idle for at least maxIdleMillisecs milliseconds.
- boolean isExpired() indicates the expiration status of the Connection.

ConnectionRegistryImpl

This is an implementation of the ConnectionRegistry interface that serves as a global repository of connections opened by ResultSetDataListImpl objects. The global nature of the repository makes it suitable for implementation as a singleton. A ResultSetDataListImpl object calls registerConnection after it opens a connection and unregisterConnection after it closes a connection.

- void touchConnection(Connection conn) calls the touch method on the CachedConnection object that contains a database connection.
- void expireIdleConnections() calls the expireIfIdle method of all CachedConnection objects in the registry. It then removes each expired CachedConnection object from the repository.

Typically, we would schedule a background process to periodically invoke the expireIdleConnections method of the singleton ConnectionRegistryImpl object.

ResultSetDataList

The following example methods have been added to this interface.

- void touchConnection() updates the timestamp associated with the underlying database connection in the ConnectionRegistry.
- void setConnectionRegistry(ConnectionRegistry registry) sets the ConnectionRegistry to be used by the ResultSetDataList to store database connections.

ResultSetDataListImpl

This class has been updated to implement the new methods of the ResultSetDataList interface. It is passed a ConnectionRegistry either during instantiation or via the setConnectionRegistry method and retains a reference to it. It stores its database connection in the ConnectionRegistry. It is passed a ResultSetContextManager instance during instantiation.

It also implements a new method:

- void resetContext(Connection conn, Statement stmt, ResultSet rs) resets its internal references to the Connection object, the underlying query Statement object, and the ResultSet object. It also registers the Connection object in the connection registry and positions the ResultSet cursor to its original location.

The implementation of touchConnection() delegates calls to the method of the same name in the ConnectionRegistry object. The ConnectionRegistry.touchConnection method throws an exception if the connection has expired. The ResultSetDataListImpl.touchConnection method handles the exception by calling the resetContext method of a ResultSetContextManagerImpl object to create a new database connection and execute the original query. The ResultSetContextManagerImpl object then calls the resetContext method of the ResultSetDataListImpl object passing it the new database connection and ResultSet object. The resetContext method of the ResultSetDataListImpl object registers the new Connection in the ConnectionRegistry and stores references to the new Connection object and ResultSet object. It then positions the cursor of the ResultSet object to its original location. At this point, everything is restored to its original state and other than noticing a slight delay the caller has no idea that anything unusual has happened.

If the data accessed by your query is volatile, the result set returned by the query when it is re-executed might be different than the original result set. In general, it would be very difficult to detect that the result set from a subsequent execution differs from the first. Only in the situation where the new result set is smaller than the original and the original result set was positioned beyond the last item of the new result set would it be easy to detect a difference. The best that can be done to get around this problem is to return a warning to the caller indicating that the result set was recreated and that it might have changed.

ResultSetContextManager

This interface is a key component of the connection expiration strategy. It knows how to restore a ResultSetDataListImpl object whose database Connection object has been expired to its original state. This interface allows us to close underlying connections while allowing an overlying ResultSetDataListImpl object to hide this from a caller. It has the following example method:

- void resetContext(ResultSetDataListImpl rs) restores the state of a ResultSetDataListImpl object whose underlying database connection has expired.

The ResultSetContextManager interface is needed because ResultSetDataListImpl is a generic class with no specific knowledge of the underlying database access logic. This knowledge is encapsulated in the application specific DataAccessObject that creates the ResultSetDataListImpl object. When a ResultSetDataListImpl object discovers through its touchConnection method that the underlying database connection has expired it needs a way to communicate with the original DataAccessObject to create a new connection and re-execute the original query. Because a DataAccessObject does not expose a generic interface, the ResultSetContextManager was introduced to serve this purpose.

ResultSetContextManagerImpl

This class implements the ResultSetContextManager interface. It accepts a DataAccessObject in its constructor and keeps a reference to it. It implements the resetContext method by requesting the DataAccessObject to create a new database connection and to execute the original query. The ResultSetContextManagerImpl object then issues a callback to the resetContext method of the ResultSetDataListImpl object parameter passing it the new Connection and ResultSet object. The resetContext method of the ResultSetDataListImpl object completes the restoration of the context by positioning the underlying ResultSet cursor to its pre-expired location.

Because this class appears to simply serve as a wrapper for the DataAccessObject class, one might wonder why we don't simply have the DataAccessObject implement the ResultSetContextManager interface directly and eliminate the need for a new class. We choose to have a separate class because we need a place to store query context information such as query

input parameter values. Without a separate class we would have to store query input values in the `DataAccessObject` itself. By not storing query state in the `DataAccessObject` we have the flexibility of implementing the `DataAccessObject` as a singleton.

ResultSetDataListChunk

Up until now we have focused on the importance of expiring connections that are idle. Now we wish to focus on preventing connections from expiring prematurely. From our prior discussion it should be clear that we call the `touchConnection` method of the `ConnectionRegistry` object to keep connections alive, but we haven't decided where and how often to call the `touchConnection` method. On the one hand we don't want to do it too often because this would introduce unnecessary call overhead. On the other hand, we need to do it frequently enough to prevent the connection from expiring prematurely.

Ideally, we would like to call `touchConnection` everytime a client initiates an action on the `DataListHandlerImpl` object, such as whenever a client calls `getDataListChunk`. Because `getDataListChunk` creates a new `ResultSetDataListChunk` object everytime it is called, we accomplish this by having the constructor of `ResultSetDataListChunk` call `touchConnection` on the underlying `ResultSetDataListImpl` object.

DataAccessObject

This class now passes an instance of a `ResultSetContextManagerImpl` instance to the constructor of the `ResultSetDataListImpl`. The `ResultSetContextManagerImpl` accepts a reference to a `DataAccessObject` in its constructor and retains it for later use as described above.

DataListHandlerImpl

This class is an implementation of the `DataListHandler` interface that accepts a `ConnectionRegistry` object as input to its constructor. It passes the `ConnectionRegistry` object to the `ResultSetDataListImpl` object that it creates to store the query search results.

Servlet Connection Expiration Strategy

In this section we describe how to apply the connection expiration strategy to a Servlet environment. This is the most efficient application of the strategy because it minimizes the number of tiers involved in moving data from the database to the client. Since search functions typically do not execute in the context of a transaction nor do they incorporate a lot of business logic, an EJB tier is not essential.

The first issue to resolve is to determine where to store the singleton `ConnectionRegistryImpl` object. Of the possible candidates, the `ServletContext` object is the logical choice because it is a persistent globally accessible repository of objects. Because it is globally accessible, a batch process can be written to periodically access the stored `ConnectionRegistryImpl` object and invoke its `expireIdleConnections` method. The other item to resolve is where to store the `DataListHandlerImpl` object across Http requests. The logical choice here is the `HttpSession` object because the `DataListHandlerImpl` object is specific to each session.

The structure of the strategy is given in figure 7 and sequence diagrams are given in figures 8 and 9.

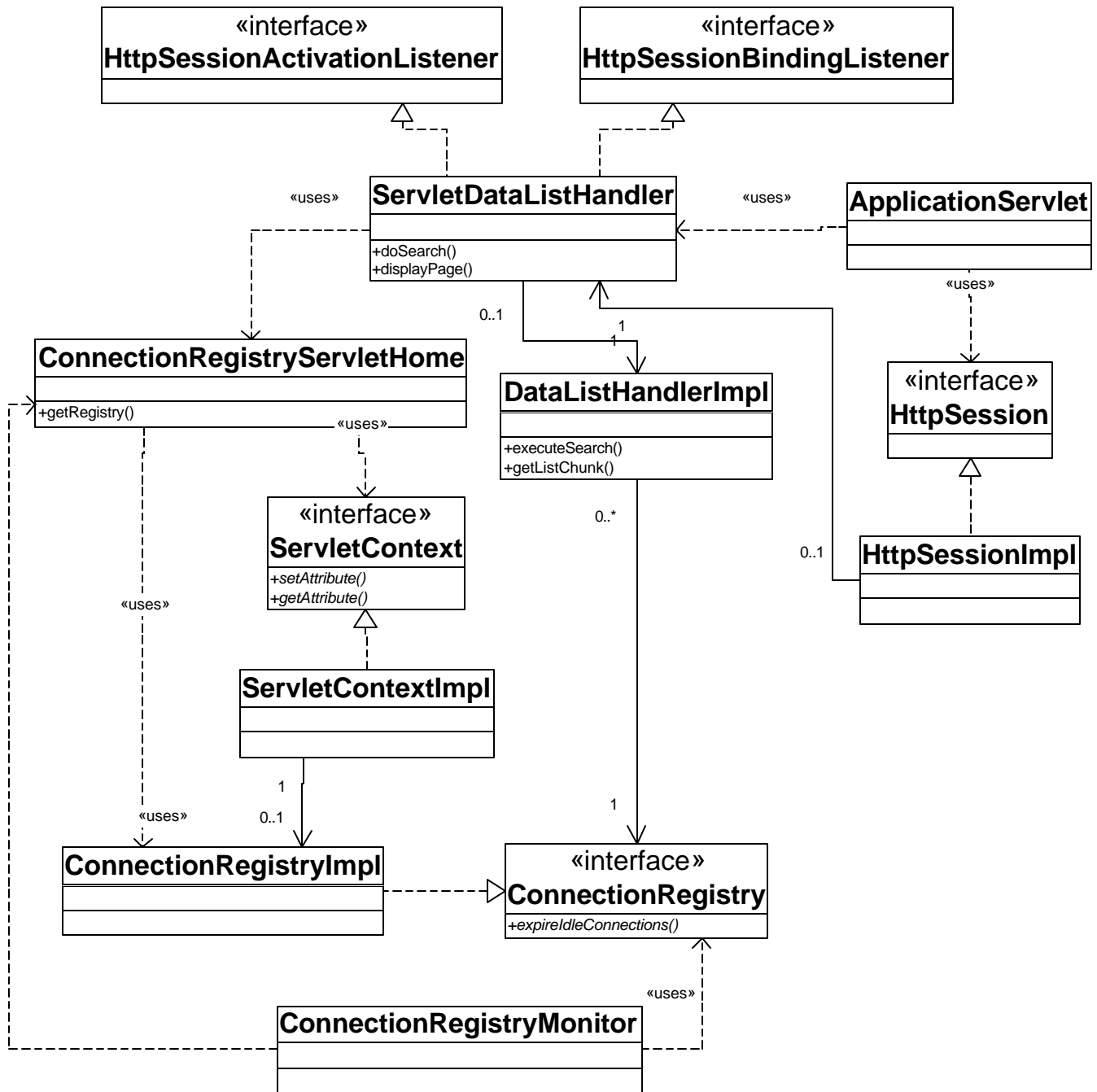


Figure 7. Servlet Connection Expiration Strategy Class Diagram

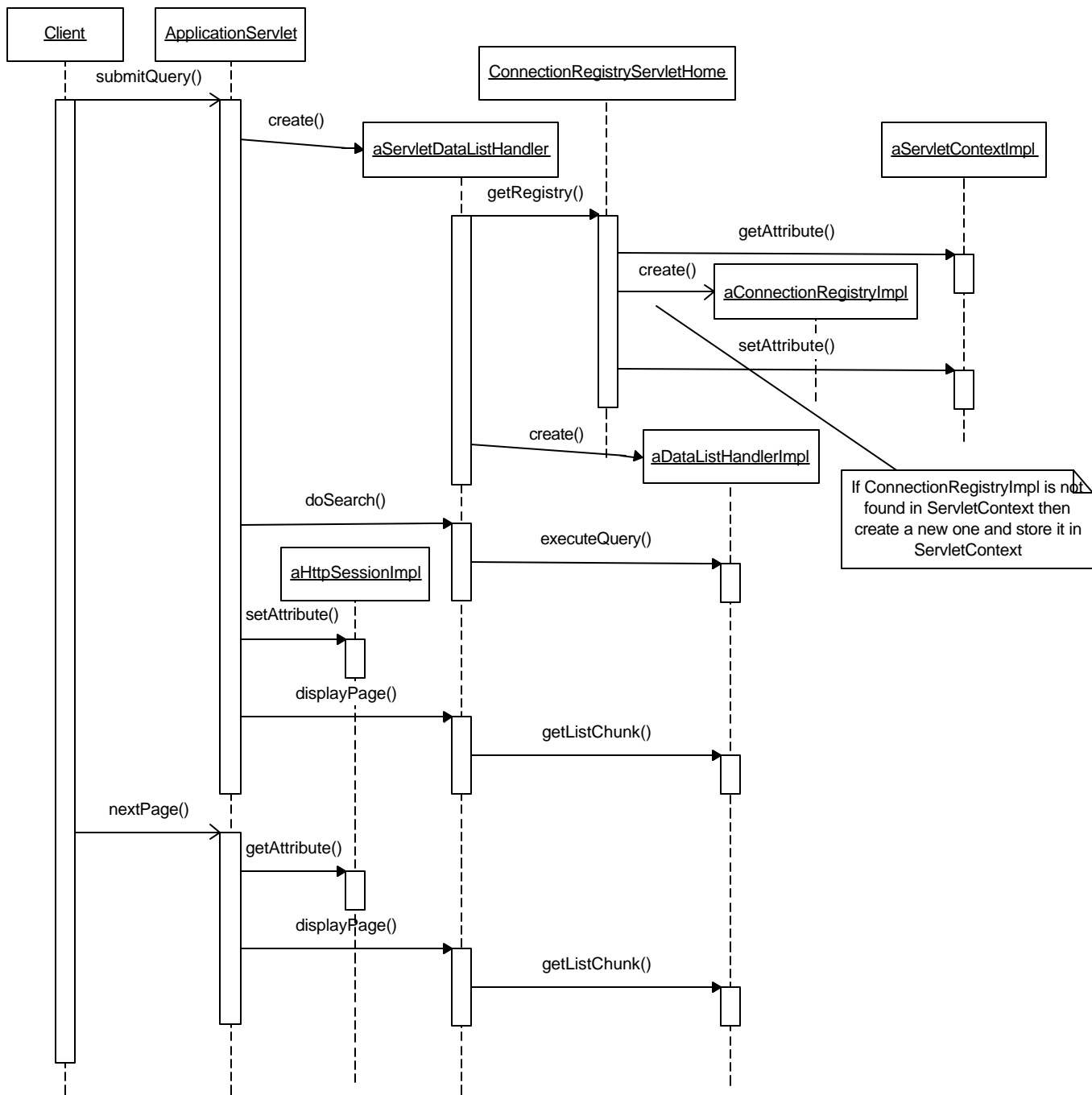


Figure 8. Servlet Connection Expiration Strategy Sequence Diagram

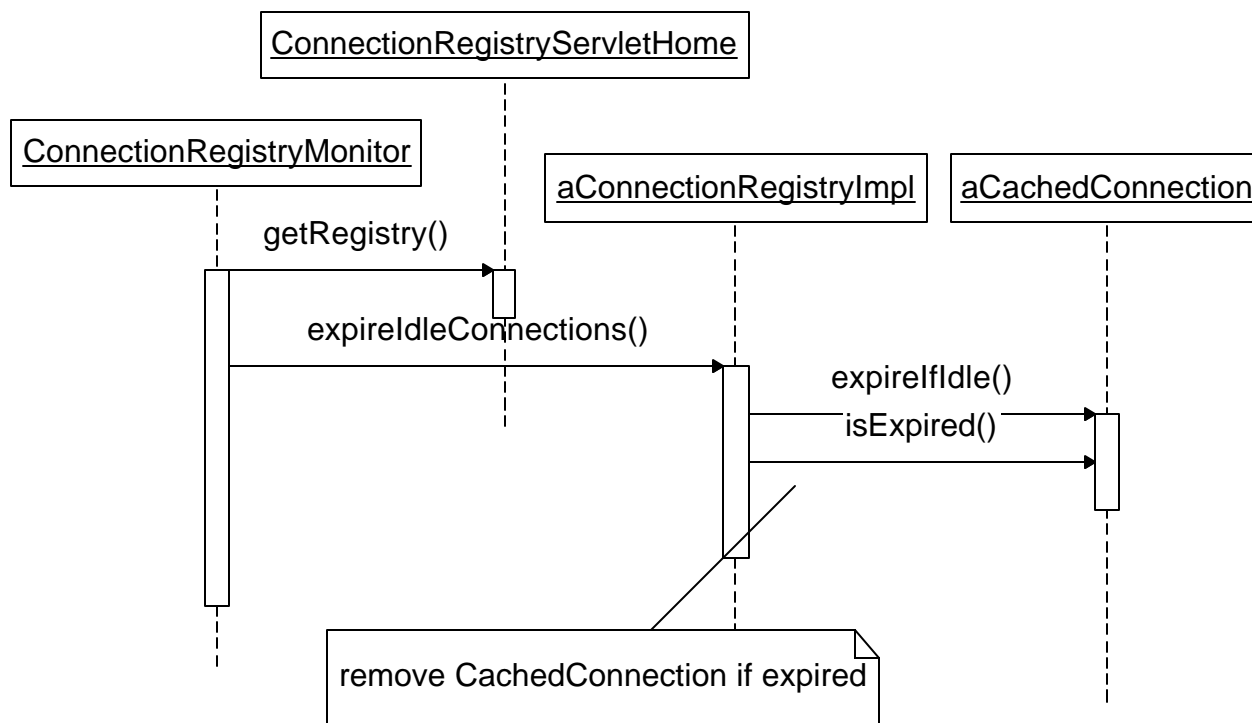


Figure 9. Servlet Connection Monitor Sequence Diagram

Strategy Participants and Collaborations

ApplicationServlet

This is a Servlet that acts as a client of the Data List Handler. The client asks the Data List Handler to execute a search and then repeatedly requests a subset of the search results. The Servlet asks a ServletDataListHandler object which is a proxy for a DataListHandlerImpl object to execute a search and then repeatedly display a page of search results.

ConnectionRegistryServletHome

In a Servlet environment we store a single ConnectionRegistryImpl instance in the ServletContext object. ConnectionRegistryServletHome is a factory class that provides access to the singleton ConnectionRegistryImpl instance. It implements the following static method:

- `getRegistry(ServletContext sc)` retrieves the singleton ConnectionRegistryImpl instance from the ServletContext parameter. If it doesn't already exist, it first creates it.

In a clustered application server environment failover of HttpSession objects to another JVM in the cluster is possible. However, the contents of the ServletContext object including the ConnectionRegistryImpl and all of its Connection objects are not failed over. Thus, the first time `getRegistry` is called after failover to a new JVM, a new ConnectionRegistryImpl instance is created and stored in the new JVM's ServletContext.

Even if it were somehow possible to failover the contents of a ServletContext, we would still not failover the

ConnectionRegistryImpl object because failover is only possible for serializable objects. Because JDBC Connection objects are not serializable, there would be no point in trying to failover the ConnectionRegistryImpl object.

ServletDataListHandler

In a Servlet environment we store the DataListHandlerImpl object in an HttpSession object in order to preserve it across Http requests. Objects stored in the HttpSession object can implement two interfaces in order to be automatically notified of important Servlet tier events, HttpSessionBindingListener and HttpSessionActivationListener. The ServletDataListHandler class was introduced to implement these interfaces as a proxy for the DataListHandlerImpl class.

The HttpSessionBindingListener.valueUnbound method is called when an object is removed from the HttpSession object, most likely in the event of a session timeout. The ServletDataListHandler implementation of this method calls the close method of its DataListHandlerImpl object. The close method calls the close method of an underlying ResultSetDataListImpl object that in turn closes its database connection.

In a clustered application server environment, it is possible to failover an active HttpSession to another JVM in the cluster. In order for a session object to fail over it must be serializable. For this reason, the ServletDataListHandler, its underlying DataListHandlerImpl and most of the objects it references directly and indirectly implement the Serializable interface. Because JDBC Connection objects are not serializable, any instance variables that reference either a JDBC Connection object or a ConnectionRegistryImpl object are declared transient. We work around the fact that Connection objects are not serializable by having the ServletDataListHandler class implement the HttpSessionActivationListener.sessionDidActivate method. This method is called during migration of the serializable contents of an HttpSession object to another JVM in the cluster. The method gets an instance of the global ConnectionRegistryImpl object from the ServletConnectionRegistryHome in the new JVM and passes it to the underlying DataListHandlerImpl object via the setConnectionRegistry method. The setConnectionRegistry method stores a reference to the new ConnectionRegistry object in a transient instance variable.

Of course, immediately after failover to a new JVM, the global ConnectionRegistryImpl object in the new JVM is empty. This means that the first time we call a method on a failed over ResultSetDataListImpl object there will be no JDBC Connection object available. This is handled the same way recovery from an expired Connection is handled in the non-failover case. The getListChunk method of the DataListHandlerImpl object instantiates a new ResultSetDataListChunk object. Its constructor gets an exception when trying to touch the non-existent JDBC Connection in the ConnectionRegistry. It then automatically reestablishes a new JDBC Connection and performs the previously documented recovery steps to restore the ResultSetDataListImpl object to its pre-failover state. This includes placing the new Connection in the ConnectionRegistryImpl object, re-executing the original query, creating a new JDBC ResultSet, and positioning the ResultSet cursor to its original location.

ConnectionRegistryMonitor

This class is a Servlet that calls the ConnectionRegistryServletHome.getRegistry method to access the singleton ConnectionRegistryImpl instance in the ServletContext. It then calls the expireIdleConnections method of the ConnectionRegistryImpl instance to close and remove any Connections that have been idle for too long. A batch job could be set up to send an Http request to this Servlet on a periodic basis.

EJB Tier Strategy

This is a variation of the relational database access strategy in which the participant classes are located in the EJB tier. In this scenario, a stateful EJB is the client of the Data List Handler API.

Distributed Servlet/EJB Tier Strategy

This is a variation of the relational database access strategy in which the participant classes are located in both the Servlet and EJB tiers. In this strategy a stateful EJB is the client of a relational database access implementation of the Data List Handler interfaces that resides in the EJB tier. A separate non-database implementation of the Data List interfaces exists in the Servlet tier. Its client is the presentation layer of the Servlet tier. The Servlet tier implementation consists of a business object delegate class that communicates with the EJB tier. A request to the Servlet tier `DataListHandlerImpl.getListChunk` method is handled by forwarding the request to the EJB tier. The EJB tier gets a `ResultSetDataListChunk` instance from its `DataListHandlerImpl` object. It then fetches the elements from the `ResultSetDataListChunk` and places them in a physical List object. It then returns the List object to the Servlet tier `DataListHandlerImpl.getListChunk` method as a `DataList` object. The Servlet tier `DataListHandlerImpl.getListChunk` method then returns the `DataList` object to the original client.

Database Connection Pooling Strategy

An additional way to minimize the impact of long-lived open connections is to take advantage of features available in the underlying database implementation. Oracle, for example, has features that substantially reduce the overhead associated with individual database connections. These features are: shared server, database server connection pooling (not to be confused with application server connection pooling), and session multiplexing. Shared server minimizes the memory footprint of a database connection on the database server machine. The other two features, server connection pooling and session multiplexing reduces the network overhead by sharing physical network connections to the database server machine among multiple connected client processes. In the newest release, 9i, these features allow Oracle to support 1000's of concurrent database connections on even a moderately configured server.

Call Level Middle-Tier Connection Pooling Strategy

A final way to minimize the impact of long-lived open connections is to take advantage of vendor-specific features of JDBC drivers. Oracle, for example, supports call level connection pooling in its JDBC driver. Traditional middle-tier connection pooling mechanisms allocate a physical connection from the pool to satisfy a request for a logical connection. The physical connection remains allocated until the client closes the logical connection. This implies that the maximum number of concurrent logical connections is equal to the number of physical connections in the pool.

Oracle's call level connection pool, on the other hand, only allocates a physical connection from the pool for the duration of a JDBC call. Once the call is completed, the physical connection is available to service another JDBC call from either the same or different logical connection. Unlike a traditional connection pool, a call level connection pool allows the number of concurrent logical connections to be far greater than the number of physical connections. This configuration can support a large number of long-lived open logical connections with minimal overhead.

Another advantage of Oracle's call level connection pooling mechanism is that the logical connections do not have to have the same database username and password as the underlying physical connections. Oracle accomplishes this by distinguishing between a physical database connection and a logical database session. Multiple logical database sessions with possibly different usernames can be multiplexed over the same physical database connection. This feature eliminates the need to map all middle tier clients to a single shared database identity when communicating with the database. By using a database user name and password associated with the currently logged in user, the true identity of the user is readily available in the database tier. There is no need to implement specialized logic to communicate the true identity to the database tier.

Consequences

Provides a Generic Interface for Retrieving Search Results

The Data List Handler pattern provides a set of interfaces, `DataListHandler`, `DataList`, and `DataListIterator` that insulate the client from the underlying details of the search and retrieval implementation. Changes to the underlying implementation require no changes to the client code.

Does not Require Search Results to be stored in a Physical Collection Object

The nature of the `DataListIterator` and `DataList` interfaces allows for underlying implementations that do not involve a physical collection of search result objects. The `DataListIterator.next` and `DataList.get` methods require an empty object as an input argument into which an item in the collection is placed. This allows the underlying implementation the option of not instantiating search result items as physical objects in a collection. This flexibility is needed for maximum efficiency.

Improves Efficiency of Retrieving Large Search Result Sets from a Relational Database

The underlying implementation for accessing data from a relational database using the `ResultSetDataList` and `ResultSetDataListIterator` interfaces and their associated implementation classes provides for maximal efficiency. It keeps the JDBC Connection and `ResultSet` objects open across client requests and makes use of JDBC 2.0 Scrollable `ResultSets` to avoid having to fetch rows from the database until they are requested by the client. This represents a significant improvement over alternative approaches that involve either pre-fetching all matching rows or re-executing the search query every time the client requests a new subset of hits.

Overhead of Long-lived Open Database Connections

The `ResultSetDataList` strategy requires the JDBC Connection to be preserved across client requests. In a multi-tiered Internet environment this could cause the JDBC Connection to remain open and idle for extended periods of time. Having potentially large numbers of open idle database connections could incur a lot of overhead both on the application server and database server. The Connection Expiration Strategy ensures that database connections that have been idle for an extended period of time are closed. It closes idle connections while logically preserving the overlying `ResultSetDataListImpl` context. It does this by transparently creating a new database connection and JDBC `ResultSet` that seamlessly takes the place of the original `ResultSet`.

Sample Code

Consider an Internet product catalog search application where a user searches for products by entering keyword search criteria.

Implementing the Data List Handler Relational Database Access Strategy with Connection Expiration

The `ProductHandler` class implements the `DataListHandler` interface. It also implements `Serializable` so that it can be stored in an `HttpSession` object that can be migrated to another JVM to support failover in a distributed Container environment.

```
package com.adept.app;

import com.adept.util.DataList;
import com.adept.util.DataListException;
```

```
import com.adept.util.DataListHandler;
import com.adept.db.ResultSetDataList;
import com.adept.db.ResultSetDataListImpl;
import com.adept.db.ResultSetDataListChunk;
import com.adept.db.ConnectionRegistry;
import com.adept.db.InvalidConnectionException;

import java.sql.*;
import java.io.Serializable;

public class ProductHandler extends Object implements DataListHandler,
    Serializable {

    private ProductCatalogDAO productCatalog;
    private ResultSetDataListImpl productDataList;
    private transient ConnectionRegistry connRegistry;

    /**
     * ConnectionRegistry is stored in a transient variable
     * because it contains JDBC Connection objects which are not
     * serializable.
     */
    public ProductHandler(ConnectionRegistry connRegistry)
        throws ProductHandlerException {
        try {
            productCatalog = new ProductCatalogDAO();
            this.connRegistry = connRegistry;
        } catch (SQLException e) {
            throw new ProductHandlerException("Could not create DAO", e);
        }
    }

    public void setConnectionRegistry(ConnectionRegistry connRegistry) {
        this.connRegistry = connRegistry;
        if (productDataList != null)
            productDataList.setConnectionRegistry(connRegistry);
    }

    /**
     * Use data access object to execute a search and return
     * a ResultSetDataList that encapsulates a JDBC ResultSet
     */
    public void executeQuery(String keywords) throws ProductHandlerException {
        try {
            productDataList = productCatalog.executeQuery(connRegistry, keywords);
        } catch (SQLException e) {
            throw new ProductHandlerException("Error executing query", e);
        }
    }

    /**
     * Get a DataList that represents a subset of the
     * search results. Construct a ResultSetDataListChunk
     * passing it a ResultSetDataList.
     */
    public DataList getListChunk(int startIndex, int count)
        throws DataListException {
```

```
    try {
        return new ResultSetDataListChunk(productDataList, startIndex, count);
    } catch (Exception e) {
        throw new DataListException("Failed to getListChunk", e);
    }
}

public boolean elementExists(int index) throws DataListException {
    try {
        return productDataList.elementExists(index);
    } catch (SQLException e) {
        throw new DataListException("Error in elementExists" + e);
    }
}

/** Release resources, such as JDBC connection. */
public void close() throws DataListException {
    productDataList.close();
}
}
```

The `ResultSetDataListImpl` class implements `ResultSetDataList`.

```
public class ResultSetDataListImpl implements ResultSetDataList {

    private ResultSetRowMapper rowMapper;
    private Connection conn;
    private Statement stmt;
    private ResultSet rs;

    public ResultSetDataListImpl(
        ResultSetRowMapper rowMapper, Connection conn,
        Statement stmt, ResultSet rs) {
        ... // Set instance variables
    }

    public DataListIterator iterator()
        throws DataListException {
        try {
            return new ResultSetDataListIterator(this);
        } catch ... // Handle exception
    }

    public void close() throws DataListException {
        ... //close ResultSet, Statement, and Connection
    }

    /**
     * Put object state in item and return it.
     * Use rowMapper to map fields of ResultSet to
     * instance variables of item.
     * ResultSet index ranges from 1, while
     * ResultSetDataList index ranges from 0.
     */
}
```

```
public Object get(int index, Object item)
    throws DataListException, IndexOutOfBoundsException {
    try {
        if (rs.absolute(index+1)) {
            if (rs.getRow() != index + 1)
                throw new IndexOutOfBoundsException();
            return rowMapper.mapRow(rs, item);
        } else
            throw new IndexOutOfBoundsException();
    } catch ... // Handle exception
}

public boolean hasNext() throws SQLException {
    return !rs.isLast() && (rs.getRow() != 0 ||
        rs.isBeforeFirst());
}

public void beforeFirst() throws SQLException {
    rs.beforeFirst();
}

// index ranges from 0 .. num_elements.
// JDBC ResultSet index ranges from 1 ..
public boolean absolute(int index) throws SQLException {
    return rs.absolute(index + 1);
}

public boolean isEmpty() throws DataListException {
    try {
        return !rs.isBeforeFirst() && !rs.isAfterLast() &&
            rs.getRow() == 0;
    } catch ... // Handle exception
}

/*
 * Preserve current location of cursor.
 * Find out if element exists
 * Put cursor back where it was to begin with.
 */
public boolean elementExists(int index)
    throws SQLException {
    boolean beforeFirst = rs.isBeforeFirst();
    boolean afterLast = rs.isAfterLast();
    int currIndex = rs.getRow();
    boolean exists = rs.absolute(index + 1);
    if (beforeFirst)
        rs.beforeFirst();
    else if (afterLast)
        rs.afterLast();
    else if (currIndex != 0)
        rs.absolute(currIndex);
    return exists;
}
}
```

The `ResultSetDataListChunk` class implements `ResultSetDataList`. It implements `Serializable` so that it can be migrated to another JVM in a distributed environment. It stores non-serializable objects like a JDBC Connection and `ResultSet` in transient variables.

```
package com.adept.db;

import com.adept.util.*;
import java.sql.*;
import java.util.*;
import java.io.Serializable;

public class ResultSetDataListImpl implements ResultSetDataList,
    Serializable {

    private ResultSetRowMapper rowMapper;
    private transient Connection conn;
    private transient Statement stmt;
    private transient ResultSet rs;
    private transient ConnectionRegistry connRegistry;
    private ResultSetContextManager rsContextManager;
    private int currIndex = -1;

    /**
     * Register the Connection object in the ConnectionRegistry so that it
     * can be expired if it is idle too long.
     */
    public ResultSetDataListImpl(ResultSetRowMapper rowMapper,
        Connection conn, Statement stmt, ResultSet rs,
        ConnectionRegistry connRegistry,
        ResultSetContextManager rsContextManager) {
        this.rowMapper = rowMapper;
        this.conn = conn;
        this.stmt = stmt;
        this.rs = rs;
        this.connRegistry = connRegistry;
        this.rsContextManager = rsContextManager;
        this.connRegistry.registerConnection(conn);
    }

    public void setConnectionRegistry(ConnectionRegistry connRegistry) {
        this.connRegistry = connRegistry;
    }

    /**
     * Updates timestamp of ConnectionRegistry to indicate that Connection is
     * active and should not be expired any time soon. If the Connection has already
     * expired, touchConnection returns an InvalidConnectionException. In this case
     * we ask the ResultSetContextManager to restore the ResultSetDataListImpl to its
     * original state. The ResultSetContextManager will create a new Connection
     * and execute the original query. It will then do a call back to the
     * the resetContext method of the ResultSetDataListImpl object to complete
     * restoration of state.
     */
    public void touchConnection() throws ResultSetDataListContextLostException,
        InvalidConnectionRegistryException, SQLException {
        if (connRegistry == null) {
```

```
        throw new InvalidConnectionRegistryException();
    }
    try {
        connRegistry.touchConnection(conn);
        return;
    } catch (InvalidConnectionException e) {
    }
    rsContextManager.resetContext(this);
}

/**
 * This method would be called by a ResultSetContextManager to complete
 * restoration of state after an idle JDBC Connection was expired.
 * This method is called after a new JDBC Connection is established and
 * the query is re-executed. This method positions the ResultSet
 * cursor to its original location.
 */
public void resetContext(Connection conn, Statement stmt, ResultSet rs)
    throws ResultSetDataListContextLostException {
    this.conn = conn;
    this.stmt = stmt;
    this.rs = rs;
    // Register the new connection
    this.connRegistry.registerConnection(conn);
    /*
     * We try to position the ResultSet to where it was before the
     * underlying Connection expired. If we can't we throw an exception.
     * This method is called after re-executing the original query. If the
     * underlying data has changed, then it is possible that the query
     * might return fewer items this time around and we might not be able
     * to position the ResultSet to the original record number.
     */
    if (currIndex == -1)
        return;
    else {
        try {
            if (rs.absolute(currIndex+1))
                return;
            throw new ResultSetDataListContextLostException();
        } catch (SQLException e) {
            throw new ResultSetDataListContextLostException(e);
        }
    }
}

public DataListIterator iterator() throws DataListException {
    try {
        return new ResultSetDataListIterator(this);
    } catch (SQLException e) {
        throw new DataListException("Error creating iterator", e);
    }
}

private void closeException(SQLException e) throws DataListException {
    throw new DataListException("Error in close" + e);
}
```

```
public void close() throws DataListException {
    if (conn != null)
        try {
            if (conn.isClosed())
                return;
        } catch (SQLException e0) {
            cleanup();
            closeException(e0);
        }
    if (rs != null)
        try {
            rs.close();
        } catch (SQLException e1) {
            cleanup();
            closeException(e1);
        }
    if (stmt != null)
        try {
            stmt.close();
        } catch (SQLException e2) {
            cleanup();
            closeException(e2);
        }
    if (conn != null)
        try {
            conn.close();
            connRegistry.unregisterConnection(conn);
        } catch (SQLException e3) {
            cleanup();
            closeException(e3);
        }
}

private void cleanup() {
    if (rs != null)
        try {
            rs.close();
        } catch (SQLException e0) {}
    if (stmt != null)
        try {
            stmt.close();
        } catch (SQLException e1) {}
    if (conn != null)
        try {
            conn.close();
            connRegistry.unregisterConnection(conn);
        } catch (SQLException e2) {}
}

public Object get(int index, Object item) throws DataListException,
    IndexOutOfBoundsException {
    try {
        if (rs.absolute(index+1)) {
            currIndex = index;
            if (rs.getRow() != index + 1)
                throw new IndexOutOfBoundsException();
            return rowMapper.mapRow(rs, item);
        }
    }
}
```

```
        } else
            throw new IndexOutOfBoundsException();
    } catch (SQLException e) {
        throw new DataListException("Error in get", e);
    }
}

public boolean hasNext() throws SQLException {
    return !rs.isLast() && (rs.getRow() != 0 ||
        rs.isBeforeFirst());
}

public void beforeFirst() throws SQLException {
    rs.beforeFirst();
}

/**
 * @param index Range is 0 .. num_elements. Result set index
 *             ranges from 1 ..
 */
public boolean absolute(int index) throws SQLException {
    if (rs.absolute(index+1)) {
        currIndex = index;
        return true;
    } else
        return false;
}

public boolean isEmpty() throws DataListException {
    try {
        return !rs.isBeforeFirst() && !rs.isAfterLast() &&
            rs.getRow() == 0;
    } catch (SQLException e) {
        throw new DataListException("Error in is Empty", e);
    }
}

public boolean hasMore() {
    return false;
}

public boolean elementExists(int index) throws SQLException {
    boolean beforeFirst = rs.isBeforeFirst();
    boolean afterLast = rs.isAfterLast();
    int currIndex = rs.getRow();
    boolean exists = rs.absolute(index + 1);
    if (beforeFirst)
        rs.beforeFirst();
    else if (afterLast)
        rs.afterLast();
    else if (currIndex != 0)
        rs.absolute(currIndex);
    return exists;
}
}
```

ResultSetDataListIterator implements DataListIterator.

```
package com.adept.db;

import com.adept.util.*;
import java.sql.*;
import java.util.*;

public class ResultSetDataListIterator
    implements DataListIterator {

    private ResultSetDataList rsDataList;
    private int currentIndex = 0;

    public ResultSetDataListIterator(ResultSetDataList rsDataList)
        throws SQLException {
        this.rsDataList = rsDataList;
        rsDataList.beforeFirst();
    }

    public boolean hasNext() throws DataListException {
        try {
            return rsDataList.hasNext();
        } catch (SQLException e) {
            throw new DataListException("Error in has next", e);
        }
    }

    public Object next(Object obj) throws NoSuchElementException,
        DataListException {
        Object object;
        try {
            object = rsDataList.get(currentIndex, obj);
        } catch (IndexOutOfBoundsException e) {
            throw new NoSuchElementException();
        }
        // Only increment index after getRow is successful.
        currentIndex++;
        return object;
    }
}
```

ProductRowMapper implements ResultSetRowMapper. It populates a ProductDataItem from the fields of a ResultSet row.

```
public class ProductRowMapper implements ResultSetRowMapper {

    public ProductRowMapper() {
    }

    public Object mapRow(ResultSet rs, Object itemObj)
        throws SQLException {
        ProductDataItem item = (ProductDataItem)itemObj;
        item.id = rs.getInt("ID");
    }
}
```

```

        item.descr = rs.getString("DESCR");
        return item;
    }
}

```

ProductCatalogRSContextManager knows how to restore the state of a ResultSetDataListImpl object after its JDBC Connection was closed for being idle for too long. It accepts a DAO in its constructor which it delegates its work to. It implements Serializable so that it can be migrated to another JVM in a distributed environment.

```

package com.adept.app;

import java.io.Serializable;
import com.adept.db.*;
import java.sql.*;

public class ProductCatalogRSContextManager implements ResultSetContextManager,
    Serializable {

    private ProductCatalogDAO dao;
    private String keywords;

    /**
     * Save the DAO and the original query search string for use
     * in resetContext.
     */
    public ProductCatalogRSContextManager(ProductCatalogDAO dao,
        String keywords) {
        this.dao = dao;
        this.keywords = keywords;
    }

    /**
     * The DAO executes the original query using the original search string.
     * Then pass the new Connection and ResultSet to
     * ResultSetDataListImpl which completes the restoration of its state.
     */
    public void resetContext(ResultSetDataListImpl rsDataList) throws SQLException,
        ResultSetDataListContextLostException {
        DBResultSet dbRs = dao.executeQuery(keywords);
        rsDataList.resetContext(dbRs.conn, dbRs.stmt, dbRs.rs);
    }
}

```

The ProductCatalogDAO is the data access object used to execute the database search and to wrap a ResultSetDataListImpl object around a JDBC ResultSet. It implements Serializable so that it can be migrated to another JVM in a distributed environment.

```

package com.adept.app;

import com.adept.db.ResultSetDataList;
import com.adept.db.ResultSetDataListImpl;
import com.adept.db.ResultSetContextManager;
import com.adept.db.ConnectionRegistry;

```

```
import com.adept.db.DBResultSet;

import java.io.Serializable;
import java.util.*;
import java.sql.*;
import java.io.Serializable;

public class ProductCatalogDAO implements Serializable {

    private ProductRowMapper productRowMapper;

    /**
     * Create ResultSetRowMapper instance to be used to map
     * rows to ProductDataItem object instances.
     */
    public ProductCatalogDAO() throws SQLException {
        productRowMapper = new ProductRowMapper();
    }

    /**
     * Execute search using keywords as search criteria.
     */
    public ResultSetDataListImpl executeQuery(ConnectionRegistry connRegistry,
        String keywords) throws SQLException {
        DBResultSet dbRs = execQuery(keywords);
        return new ResultSetDataListImpl(
            productRowMapper, dbRs.conn, dbRs.stmt, dbRs.rs, connRegistry,
            new ProductCatalogRSContextManager(this, keywords));
    }

    /**
     * DBResultSet is a convenience class for storing
     * a JDBC Connection, Statement and ResultSet object.
     */
    DBResultSet execQuery(String keywords)
        throws SQLException {
        Connection conn = null;
        PreparedStatement stmt = null;
        ResultSet rs = null;
        try {
            conn = getConnection();
            stmt = conn.prepareStatement(
                "select id, descr from product " +
                "where descr like '%'||upper(?)||'%' order by id",
                ResultSet.TYPE_SCROLL_INSENSITIVE,ResultSet.CONCUR_READ_ONLY);
            stmt.setString(1,keywords);
            rs = stmt.executeQuery();
            return new DBResultSet(conn,stmt,rs);
        } catch (SQLException se) {
            closeResultSet(rs);
            closeStatement(stmt);
            closeConnection(conn);
            throw se;
        }
    }
    ...
}
```

```
}
```

ConnectionRegistryImpl is a repository of CachedConnection objects that implements the ConnectionRegistry interface.

```
package com.adept.db;

import java.util.*;
import java.sql.*;

public class ConnectionRegistryImpl implements ConnectionRegistry {

    private HashMap cachedConnections = new HashMap();
    private long maxConnectionIdleMillisecs;

    /**
     * Sessions are expired if they are idle for > maxConnectionIdleSecs
     * seconds.
     */
    public ConnectionRegistryImpl(int maxConnectionIdleSecs) {
        this.maxConnectionIdleMillisecs = maxConnectionIdleSecs * 1000;
    }

    public void registerConnection(Connection conn) {
        cachedConnections.put(conn,new CachedConnection(conn));
    }

    public void touchConnection(Connection conn) throws InvalidConnectionException {
        CachedConnection cachedConn = (CachedConnection)cachedConnections.get(conn);
        if (cachedConn == null) {
            throw new InvalidConnectionException();
        }
        cachedConn.touch();
    }

    public void unRegisterConnection(Connection conn) {
        cachedConnections.remove(conn);
    }

    /**
     * Expire any connections that have been idle for > maxConnectionIdleSecs
     * seconds. Because we are operating on a single data structure on which
     * we perform multiple discrete operations, we must synchronize access.
     * This should not create a bottleneck because there is no urgency
     * to expiring connections.
     */
    public synchronized void expireIdleConnections() {
        Iterator it = cachedConnections.values().iterator();
        while (it.hasNext()) {
            CachedConnection cachedConn = (CachedConnection)it.next();
            cachedConn.expireIfIdle(maxConnectionIdleMillisecs);
            if (cachedConn.isExpired()) {
                it.remove();
            }
        }
    }
}
```

CachedConnection is a container class for a JDBC Connection object stored in the ConnectionRegistry. It provides lastTouchedTime and expired instance variables. The lastTouchedTime is used to identify Connections that have been idle for too long and need to be expired.

```
package com.adept.db;

import java.sql.Connection;
import java.util.Date;
import java.sql.*;

public class CachedConnection {
    private Connection conn;
    private Date lastTouchedTime = new Date();
    private boolean expired = false;

    CachedConnection(Connection conn) {
        this.conn = conn;
    }

    /**
     * Updates lastTouchedTime to indicate that Connection is active. Method
     * is synchronized because we want to avoid situation where one thread
     * calls this method at about the same time that a Connection Monitor
     * thread calls expireIfIdle on the same CachedConnection object. In this case,
     * the touch could succeed in updating the lastTouchedTime, but before the
     * method completes expireIfIdle in another thread could have closed the
     * Connection.
     */
    public synchronized void touch() throws InvalidConnectionException {
        if (!expired)
            lastTouchedTime = new Date();
        else
            throw new InvalidConnectionException();
        if (!JdbcUtil.isValidConnection(conn)) {
            try {
                conn.close();
            } catch (SQLException e) {
            }
            expired = true;
            throw new InvalidConnectionException();
        }
    }

    /**
     * If the lastTouchedTime indicates that Connection has been idle longer
     * than maxIdleMillisecs, then close the Connection and set expired to true.
     * We use synchronization in order to avoid a race condition between
     * this method and touch in different threads.
     */
    public void expireIfIdle(long maxIdleMillisecs) {
        long now = new Date().getTime();
        synchronized(this) {
            if (lastTouchedTime.getTime() + maxIdleMillisecs >= now)
                return;
            try {
```

```

        conn.close();
        expired = true;
    } catch (SQLException e) {
    }
}

public boolean isExpired() {
    return expired;
}
}

```

Implementing the Servlet Connection Expiration Strategy

The interaction of a client servlet program with the the `DataListHandler` interface is illustrated. The `doGet` method presents a form that allows a user to enter search criteria. The `doPost` accepts a request either to execute a new search or to scroll to a new page of results for a search in progress. It delegates most of its work to a `ServletProductHandler` object which is a proxy for `ProductHandler` which implements the `DataListHandler` interface.

```

package com.adept.client;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class DataListClientServlet1 extends HttpServlet {
    private static final String CONTENT_TYPE = "text/html; charset=windows-1252";
    private static final String NEXT_PAGE_FIELD = "nextPage";
    private static final String PREV_PAGE_FIELD = "prevPage";
    private static final String NEXT_PAGE_LABEL = "Next";
    private static final String PREV_PAGE_LABEL = "Previous";
    private static final String PRODUCT_SEARCH_HANDLER = "PRODUCT_SEARCH_HANDLER";
    private static final String ACTION = "action";
    private static final String QUERY_FIELD = "queryString";
    private static final String SUBMIT_QUERY = "SUBMIT_QUERY";
    private static final String NAVIGATE_PAGE = "NAVIGATE_PAGE";
    private static final String START_INDEX = "startIndex";
    private static final String END_INDEX = "endIndex";
    private static final int PAGE_SIZE = 10;

    //ServletProductHandler phandler; Debugging

    public void init(ServletConfig config) throws ServletException {
        super.init(config);
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        response.setContentType(CONTENT_TYPE);
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head><title>Enter Query</title></head>");
        out.println("<body><br><br>");
        out.println("<br><br><form>");

```

```

    out.println(" method=\"post\"");
    out.println(" action=\"\" + request.getRequestURL() + "\">");
    out.println(" <input type=\"hidden\" name=\"\" + ACTION + "\"" value=\"\" +
        SUBMIT_QUERY + "\">");
    out.println(" <input type=\"text\" name=\"\" + QUERY_FIELD + "\">");
    out.println(" <input type=\"hidden\" name=\"startIndex\" value=\"-1\">");
    out.println(" <input type=\"hidden\" name=\"endIndex\" value=\"-1\">");
    out.println(" <input type=\"hidden\" name=\"\" + NEXT_PAGE_FIELD +
        "\"" value=\"true\">");
    out.println(" <input type=\"submit\" name=\"Search\" value=\"Search\">");
    out.println("</form>");
    out.println("</body></html>");
    out.close();
}

public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    String action = request.getParameter(ACTION);
    if (action.equals(SUBMIT_QUERY))
        executeQuery(request, getServletContext());
    displayQueryHitPage(request, response);
}

/*
 * Create a new ServletProductHandler to do search and store
 * it in HttpSession object for subsequent requests.
 */
private void executeQuery(HttpServletRequest request,
    ServletContext context) throws ServletException {
    try {
        String query = request.getParameter(QUERY_FIELD);
        ServletProductHandler handler = new ServletProductHandler(context);
        handler.doSearch(query);
        //phandler = handler;
        request.getSession().setAttribute(PRODUCT_SEARCH_HANDLER, handler);
    } catch (Exception e) {
        throw new ServletException(e);
    }
}

/*
 * Display a page of results using the ServletProductHandler which
 * is stored in the HttpSession object.
 */
private void displayQueryHitPage(HttpServletRequest request,
    HttpServletResponse response) throws ServletException {
    int startIndex = Integer.parseInt(request.getParameter(START_INDEX));
    int endIndex = Integer.parseInt(request.getParameter(END_INDEX));
    String prevPage = request.getParameter(PREV_PAGE_FIELD);
    int newStartIndex;
    if (prevPage != null && prevPage.equals("true"))
        newStartIndex = startIndex - PAGE_SIZE;
    else
        newStartIndex = endIndex + 1;
    //ServletProductHandler handler = phandler;
    ServletProductHandler handler = (ServletProductHandler)
        request.getSession().getAttribute(PRODUCT_SEARCH_HANDLER);
}

```

```

    try {
        handler.displayPage(request, response, CONTENT_TYPE,
            newStartIndex, PAGE_SIZE,
            PREV_PAGE_FIELD, NEXT_PAGE_FIELD, PREV_PAGE_LABEL,
            NEXT_PAGE_LABEL, ACTION, NAVIGATE_PAGE);
    } catch (Exception e) {
        throw new ServletException(e);
    }
}
}
}

```

ServletProductHandler is a proxy for ProductHandler which implements the DataListHandler interface. It is stored in the HttpSession object and implements interfaces that allow it to react to session life-cycle events. It implements Serializable to support migration to a new JVM in a distributed environment. It also handles some display functionality. Search results are stored in objects of class ProductDataItem.

```

package com.adept.client;

import com.adept.app.ProductDataItem;
import com.adept.app.ProductHandler;
import com.adept.app.ProductHandlerException;
import com.adept.util.DataList;
import com.adept.util.DataListException;
import com.adept.util.DataListIterator;
import com.adept.util.DataListHandler;
import com.adept.db.ConnectionRegistryServletHome;

import javax.servlet.*;
import javax.servlet.http.*;

import java.io.PrintWriter;
import java.io.Serializable;
import java.io.IOException;

public class ServletProductHandler extends Object
    implements HttpSessionActivationListener, HttpSessionBindingListener,
    Serializable {

    private static final String CONTENT_TYPE = "text/html; charset=windows-1252";

    ProductDataItem item = new ProductDataItem();
    ProductHandler handler;

    /**
     * Creates a new ProductHandler to which it delegates most of its
     * work. It gets a ConnectionRegistry from the
     * ConnectionRegistryServletHome.
     */
    public ServletProductHandler(ServletContext sc)
        throws ProductHandlerException {
        handler = new ProductHandler(
            ConnectionRegistryServletHome.getRegistry(sc));
    }

    public boolean doSearch(String keywords) throws ProductHandlerException,

```

```

        DataListException {
        handler.executeQuery(keywords);
        return handler.elementExists(0);
    }

public void displayPage(HttpServletRequest request,
    HttpServletResponse response, String
    contentType, int startIndex, int count, String previousFieldName,
    String nextFieldName, String previousLabel, String nextLabel,
    String actionFieldName, String action)
    throws DataListException, IOException {
    response.setContentType(contentType);
    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<head><title>Query Results</title></head>");
    out.println("<body><br><br>");
    DataList dl = handler.getListChunk(startIndex, count);
    DataListIterator iterator = dl.iterator();
    int i = startIndex;
    while (iterator.hasNext()) {
        item = (ProductDataItem)iterator.next(item);
        out.println("ID:" + item.getId());
        out.println("DESCR:" + item.getDescr() + "<BR><BR>");
        i++;
    }
    boolean moreBefore;
    if (startIndex == 0)
        moreBefore = false;
    else
        moreBefore = handler.elementExists(startIndex-1);
    boolean moreAfter = handler.elementExists(startIndex+count);
    if (moreBefore)
        printForm(request, out, startIndex,i-1, previousFieldName,
        previousLabel, actionFieldName, action);
    if (moreAfter)
        printForm(request, out, startIndex,i-1, nextFieldName,
        nextLabel, actionFieldName, action);
    out.println("</body></html>");
    out.close();
}

private void printForm(HttpServletRequest request, PrintWriter out,
    int startIndex, int endIndex, String navigateFieldName, String label,
    String actionFieldName, String action) {
    out.print("<br><br><form>");
    out.print(" method=\"post\"");
    out.println(" action=\"" + request.getRequestURL() + "\"");
    out.println(" <input type=\"hidden\" name=\"startIndex\" value=\"" +
        startIndex + "\"");
    out.println(" <input type=\"hidden\" name=\"endIndex\" value=\"" +
        endIndex + "\"");
    out.println(" <input type=\"hidden\" name=\"" + navigateFieldName +
        "\" value=\"true\"");
    out.println(" <input type=\"hidden\" name=\"" + actionFieldName +
        "\" value=\"" + action + "\"");
    out.println(" <input type=\"submit\" name=\"page\" value=\"" + label +
        "\"");
}

```

```
        out.println("</form>");
    }

    public void cleanup() throws DataListException {
        handler.close();
    }

    /**
     * Method of the HttpSessionActivationListener interface.
     * It is called when an HttpSession object is migrated to
     * a new JVM in a distributed environment to support failover.
     * Because the ConnectionRegistry and its contents are not
     * serializable, it does not migrate to the new JVM so we must
     * obtain a new ConnectionRegistry in the new JVM and pass it to the
     * migrated ProductHandler.
     */
    public void sessionDidActivate(HttpSessionEvent se) {
        ServletContext sc = se.getSession().getServletContext();
        handler.setConnectionRegistry(
            ConnectionRegistryServletHome.getRegistry(sc));
    }

    public void valueBound(HttpSessionBindingEvent event) {
    }

    /**
     * Method of the HttpSessionBindingListener interface.
     * It is called when an object is removed from the HttpSession
     * such as when the HttpSession expires. cleanup
     * ultimately closes the underlying JDBC Connection.
     */
    public void valueUnbound(HttpSessionBindingEvent event) {
        try {
            cleanup();
        } catch (DataListException e) {}
    }

    public void sessionWillPassivate(HttpSessionEvent se) {
    }
}

```

ConnectionRegistryServletHome is a factory class that creates and controls access to a singleton ConnectionRegistryImpl object that is stored in the web application's ServletContext.

```
package com.adept.db;

import javax.servlet.ServletContext;

public class ConnectionRegistryServletHome {

    private static final String CONN_REGISTRY_NAME =
        "com.adept.db.CONNECTION_REGISTRY_NAME";
    private static final String CACHED_CONNECTION_TIMEOUT_SECS =
        "com.adept.db.CACHED_CONNECTION_TIMEOUT_SECS";
}

```

```

/**
 * Retrieve the singleton ConnectionRegistryImpl object from the ServletContext.
 * If it is not found, then create a new instance and store it in the
 * ServletContext. We use synchronization because another thread could
 * have created a new ConnectionRegistryImpl instance and stored it in the
 * ServletContext immediately after the current thread checked but before it had
 * a chance to create a new instance.
 * CACHED_CONNECTION_TIMEOUT_SECS is the name of an Init Parameter for this
 * web application that indicates the idle connection time out time in seconds.
 */
public static ConnectionRegistry getRegistry(ServletContext context) {
    ConnectionRegistry registry =
        (ConnectionRegistry)context.getAttribute(CONN_REGISTRY_NAME);
    if (registry != null)
        context.log("registry found");
    else
        context.log("registry not found");
    if (registry == null) {
        synchronized(ConnectionRegistryServletHome.class) {
            registry =
                (ConnectionRegistry)context.getAttribute(CONN_REGISTRY_NAME);
            if (registry == null) {
                int maxIdleSecs =
                    Integer.parseInt(context.getInitParameter(
                        CACHED_CONNECTION_TIMEOUT_SECS));
                registry = new ConnectionRegistryImpl(maxIdleSecs);
                context.setAttribute(CONN_REGISTRY_NAME, registry);
            }
        }
    }
    return registry;
}
}

```

ConnectionRegistryMonitor is a servlet that obtains the singleton ConnectionRegistryImpl object stored in the ServletContext from ConnectionRegistryServletHome. It then calls expireIdleConnections to close and discard any JDBC connections that have been idle too long. A background job would typically be configured to periodically call this Servlet to expire idle connections.

```

package com.adept.db;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class ConnectionRegistryMonitor extends HttpServlet {
    private static final String CONTENT_TYPE = "text/html; charset=windows-1252";

    public void init(ServletConfig config) throws ServletException {
        super.init(config);
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

```

```
ConnectionRegistry cr =
    ConnectionRegistryServletHome.getRegistry(getServletContext());
cr.expireIdleConnections();
response.setContentType(CONTENT_TYPE);
PrintWriter out = response.getWriter();
out.println("<html>");
out.println("<head><title>ConnectionRegistryMonitor</title></head>");
out.println("<body>");
out.println("<p>Idle Connections have been expired.</p>");
out.println("</body></html>");
out.close();
}
```

Related Patterns

Value List Handler [CJP]

The Value List Handler pattern described in *Core J2EE Patterns* also provides an architecture for insulating the presentation layer from the details of the search and data access implementation. However, it does not address the problem of efficiency because it materializes the entire search results as a physical collection.

Iterator [DP]

Like the Value List Handler pattern, the Data List Handler pattern is based on the Iterator pattern of [DP].

Business Delegate [CJP]

In the distributed Servlet tier/EJB tier strategy, a business delegate object mediates the communication between a non-ResultSetDataList implementation of DataList in the Servlet tier and a ResultSetDataList implementation of DataList in the EJB tier.

Conclusion

The DataListHandler design pattern addresses the problem of processing Internet search queries that return large result sets in an efficient manner. It provides a set of interfaces, DataListHandler, DataList, and DataListIterator, that insulate the client from the underlying details of the search and retrieval implementation. The DataListIterator and DataList interfaces address the efficiency problem by being well suited to implementations that do not incur the overhead of instantiating the search results in a physical collection object.

A strategy for implementing the DataListHandler pattern for efficiently accessing data from a relational database that makes use of the lower level interfaces, ResultSetDataList and ResultSetDataListIterator, was described. The strategy keeps the JDBC Connection and ResultSet objects open across client requests and utilizes JDBC 2.0 Scrollable ResultSets to avoid having to fetch rows from the database until they are requested by the client. This represents a significant improvement over alternative approaches that either pre-fetch all matching rows or re-execute the search query every time the client requests a new subset of hits.

The ResultSetDataList strategy preserves the JDBC Connection across client requests. In a multi-tiered Internet

environment the JDBC Connection may remain open and idle for extended periods of time. The Connection Expiration strategy prevents the accumulation of large numbers of open connections by closing idle connections, while at the same time logically preserving the context of the overlying ResultSetDataListImpl object. The Connection Expiration strategy is also useful in a clustered application server environment where it can be used to transparently failover an active search session to another JVM in the cluster.

It must be emphasized that this pattern is most applicable when the number of search hits is large. In cases where the number of possible search hits is small or the overhead of re-executing the search is low, the inherent complexity of the DataListHandler pattern and the relational database access strategy may not justify its use. In such cases, it might be sufficient to simply store all the search results in a physical collection object prior to presenting them to the user. If all the search hits are instantiated in a physical collection, the DataListHandler pattern actually adds performance overhead because the DataListIterator.next() method copies the requested object into an empty object provided by the client. Because the item already exists as an instantiated object in the collection the copy operation is unnecessary. You can directly return the item in the collection to the client.

References

[DP] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley, 1994

[CJP] Deepak Alur, John Crupi, Dan Malks, "Core J2EE Patterns: Best Practices and Design Strategies", Sun Microsystems Press, 2001

About the Author:

Claudio Fratarcangeli has 17 years of experience using Oracle (since V4) in application development, software product development, and DBA. For the past 3 years he has been focusing on Java J2EE architecture and development. He has presented at various regional and national conferences and has authored articles in various database trade magazines and academic journals. He can be reached via email at claudiof@computer.org.