A Tuscany White Paper

Data Access Service:

How to access relational data in terms of Service Data Objects

Authors:
Kevin Williams
Brent Daniel
Version 0.1.3

Introduction/Summary

Service Data Object (SDO) has become a foundation technology for Service Oriented Architecture (SOA). Recently, BEA, IBM, Oracle, SAP, IONA, Siebel and Sybase announced their support for an SOA-enabling framework specification named Service Component Architecture (SCA). SDO provides the primary data representation within this framework.

Although not addressed by the current SDO or SCA specifications, there is a definite need for a generic data access service that operates in terms of SDOs. The alternative to this service would be the tedious and error-prone development of a custom mapping between the back-end data representation and Service Data Objects.

The Relational Database Data Access Service (RDB DAS) obviates the need for this custom development by providing a robust data access utility built around SDO. Because of its tight integration with SDO, the RDB DAS is also a perfect solution for data access within an SCA-based application.

By employing the RDB DAS, applications avoid the details and complications of working directly with a relational database and also the error prone transformation between relational rows/columns and Data Object types/properties.

Background

Since the release of the specification in late 2003, SDO has proven itself a flexible and robust technology for data representation. Its inherent support for disconnected operations and heterogeneous data sources offers strong support for the needs of modern software architectures. For these reasons, SDO has found its way into several commercial products by major vendors and these same characteristics have lead to its inclusion into SCA as a foundation technology.

SDO provides the general-case mechanism for moving data around within an SCA-enabled application. However, the reality is that most of this data must originate from some database at one edge of the application and be stored in some database at another edge. Unfortunately, database access is not currently in scope for either SDO or SCA[1].

This leaves the developer with a serious development undertaking. There is a fundamental mismatch[2] between the objects that an application works with and the tables and rows of a relational database that provide the persistent store for the object's state.

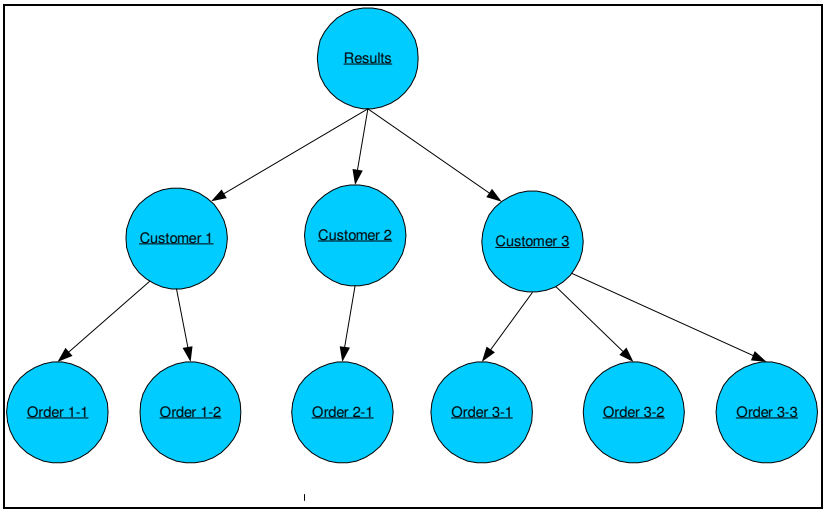
For example, let's consider a simple query against a relational database for customers in a certain age range and their related orders.

---

[1] The SDO Data Access Service (DAS) is on the SDO roadmap and will be specified in SDO 3.0

[2] See http://en.wikipedia.org/wiki/Object-Relational_Impedance_Mismatch

What the SDO-enabled application wants to work with is a normalized graph of Data Objects representing the query results that provides simple traversal between related elements. The following diagram illustrates this graph of connected Data Objects.



This in-memory graph of data objects bring to bear all of the capabilities of SDO.

- It is a disconnected representation of the queried data
- It tracks all changes from its original form via the SDO change summary
- It contains no redundant information
- It is easily serialized to XML

But unfortunately, the relational database returns a tabular representation of the query result complete with redundant Customer information as shown in the following diagram.

Customer 1	Order 1-1
Customer 1	Order 1-2
Customer 2	Order 2-1
Customer 3	Order 3-1
Customer 3	Order 3-2
Customer 3	Order 3-3

The transformation required to convert from tabular format to a graph of interconnected data objects is complicated and the reverse (transforming graph changes to a sequence of SQL inserts/updates and deletes) is even more so.

Because of the difficulties inherent in the transformation between the database and the application object space, an application development project can easily spend 20-40% of its development resources on function related to moving state in and out of the database

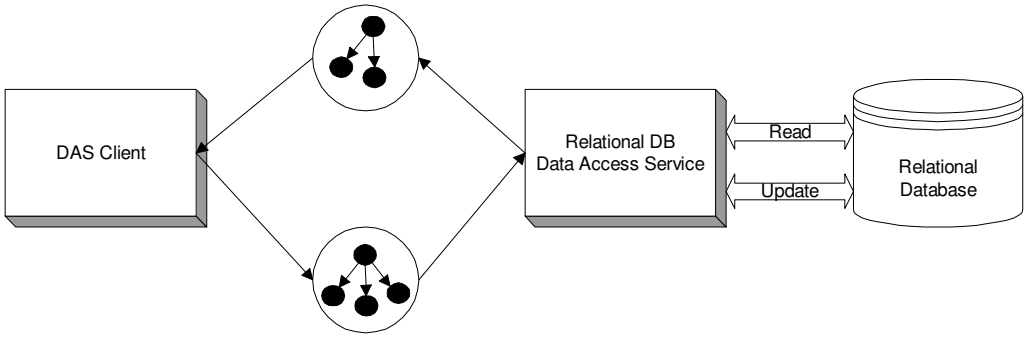
Business application developers should not be burdened with this task and should instead be allowed to focus on business functionality.

Solution

The RDB DAS offers a solution to the problems mentioned above by providing two major capabilities. The RDB DAS can:

1. Execute SQL queries and return results as a graph of Data Objects
2. Reflect changes made to a graph of Data Objects back to the database

The following diagram illustrates these two capabilities in a typical client interaction. The client starts by reading a graph of data specified by some query. The client then makes modifications to the graph, possibly by adding elements, and then requests the DAS to push the changes back to the database.



The DAS provides an intuitive interface and is designed such that simple tasks are simple to complete while more complicated tasks are just a little less simple.

The application interface to the DAS is based on the familiar Command Pattern[3]. And interaction with the DAS consists of acquiring a command instance and executing it. The following example demonstrates the simplest possible read of data.

```
String readSql = "select * from CUSTOMER where ID = 1"
Command readCustomer = Command.FACTORY.createCommand(readSql);
readCustomers.setConnection(getConnection());
DataObject root = readCustomer.executeQuery();
DataObject cust =  root.getDataObject("CUSTOMER[1]");
```

In this case the command is created programmatically from a Command factory and the only input necessary is the SQL SELECT statement.

---

[3] See Design Patterns. Gamma, Helm, Johnson, Vlissides. Addison Wesley, 1995

Pushing changes back to the database can be equally straightforward. Continuing with this example we can modify the customer object and then direct the DAS to send the modifications to the database.

```
cust.setString (“LASTNAME”, “Williams”);
ApplyChangesCommand apply = Command.FACTORY.createApplyChangesCommand();
apply.setConnection(getConnection());
apply.addPrimaryKey( "CUSTOMER.ID" );
apply.execute( root );
```

The DAS will generate the SQL update statement required to push the change back to the database. In order to enable generation of this statement we have to provide a bit of metadata to indicate the primary key of the underlying table.

Although the user can choose to build commands programmatically, the preferred approach is to define a group of commands with a configuration file. A Command Group is initialized from this configuration and provides the commands based on an assigned name. At runtime, the client asks for a command by name and executes it.

We can rework the “read” portion of the previous example to use a Command Group as follows:

```
CommandGroup cmdGroup =
        cmdGroup.FACTORY.createCommandGroup(getCustomerConfig());
cmdGroup.setConnection(getConnection());
Command readCustomer = cmdGroup.getCommand("read customer");
DataObject root = readCustomer.executeQuery();
DataObject cust =  root.getDataObject("CUSTOMER[1]”);
```

The “write” section makes use of the same command group and looks like this:

```
cust.setString (“LASTNAME”, “Williams”);
ApplyChangesCommand apply = cmdGroup.getApplyChangesCommand();
apply.execute( root );
```

As you can see, the connection is set for the group rather than for each individual command. Also, the primary key metadata has moved to the configuration file along with the SELECT string for the “get customer” command.

The configuration file for this example is an XML document and looks like this:

```
<?xml version="1.0" encoding="ASCII"?>
<Config>

   <Command name="get customer" SQL="select * from CUSTOMER where ID = 1" kind = "Select" />

   <Table name="CUSTOMER">
      <Column name="ID" primaryKey="true"/>
   </Table>
```

```
</Config>
```

The DAS supports simple database interactions as this example illustrates but also much more complex scenarios such as those requiring:

- Statically typed (generated) SDO DataObjects
- Optimistic concurrency control
- Generated database IDs
- Stored procedures
- External transaction participation
- Write-operation ordering (database constraints)
- Simple name mapping (Table/Column -> SDO Type/property)
- Column-type conversions
- Paging

Business benefits

Object to Relational Data Access

The RDB DAS provides a capable and flexible data access mechanism to applications integrating SDO technology. By employing the DAS, developers avoid developing a custom data access framework; a task that is tedious, complex and error prone.

Integrated with SDO

The Data Transfer Object pattern[4] is often used by applications to move persistent state from one part of the application architecture to another. This is especially true if the data movement requires serialization. Such an application may employ some object-to-relational technology (JDO, EJB Entity beans, etc) to retrieve the data from a backend data store and then copy the data to the DTO for transfer around the application.

The creation of separate DTOs is not necessary for an SDO-integrated application using the DAS because the SDOs themselves are easily serialized to XML.

Conclusion

The RDB DAS and SDO provide a simple and powerful way to access and work with relational data. The RDB DAS provides developers the ability to work with SDO without building custom data access solutions since the DAS works in terms of SDOs. It simplifies data access by hiding many of its complexities while still allowing developers to harness more powerful features in complex scenarios.

Because the RDB DAS is integrates SDO technology, it is a natural fit for data access within the SCA framework. In fact, the RDB DAS is evolving as part of the SOA Apache incubator project "Tuscany," and is on the roadmap for the upcoming SDO 3.0 specification.

---

[4] See http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html

More information about the RDB DAS and the implementation under development can be found here: http://incubator.apache.org/projects/tuscany