SDN Community Contribution

(This is not an official SAP document.)

Disclaimer & Liability Notice

Applies To:

Web Application Server 6.40, sp10 or greater, Java stack, BW 3.1 or greater

Summary

This is the last of a series of articles focusing on some specific design issues we faced during the development of our Java-based Custom Composite Application (CCA). These articles are rather tightly focused and are not meant to attempt to address the bigger issues of ESA and how to architect a CCA. This article focuses on the issue of how to use BW as a data source within an ESA world.

By: Richard Andrulis, Nir Paikowsky
Company: SAP
Date: 15 February 2006

Use of BW as a Data Source

One of the fundamental questions that arises in the ESA world is where should the data be stored? Or more to the point, when creating a composite application, should the data be replicated or accessed remotely? Replicating the data allows for fast access to all the data needed for your specific application; however, replication is fraught with perils to maintain data integrity. And there is the issue of the development/ maintenance effort to write the data load programs. Remote access is fine in theory but runs into trouble when you have large amounts of data or need to make several remote calls to aggregate all of the data needed for you application.
SAP's Business Information Warehouse provides a middle solution to these two options. BW allows you to aggregate your data into one central data store so you only need a single remote call to access all of your data. Because the OLAP processor is optimized for read access, BW can handle reading large amounts of data. This combined with the XML/A web service available in BW from 3.0 on, makes the access of data from BW quite an attractive option (see Table 1).

Alternative	Pros	Cons
Local Database	Easy Access Fast Response Time Independent Implementation	Requires special data loads (extractors) Requires proprietary authorization mechanism Information is not up-to date
Back-End Systems	All the information already exists Information is up-to date Simple to handle create and update	Creates extra load on back-end Requires aggregation of information from several sources
BW	Provides out-of-the-box authorization mechanism Most data is loaded to BW anyway Optimized for read-only access	Information may not be up-to date (Especially with newly created data)

Table 1: Data Source Options

Connecting to BW from the Java world

While BW has a quite robust and well-developed web front-end, to make full use of it as a data source in a composite application means that we needed to retrieve the data in a raw format that we could manipulate before displaying in our UI. To retrieve data from BW, we used the BI SDK to call the XML/A web service in

BW. There is a fair amount of documentation on MDX and XML/A so we will not go into detail here (see Summary section for links). XML/A is a web service protocol developed by Microsoft that has been adopted as standard for retrieving data. MDX is an XML schema for transferring that data through the XML/A services.

We needed to query BW for many different kinds of data. We have simple requests to return a List of Values (LOV) for Master Data or customizing entries. We had parameter queries where we needed to retrieve detailed information for a specific object. And we had quite complex queries to search for items by many (over 20) different parameters. Every service to retrieve data from BW went thru the same steps (example below is shown for LOV query):

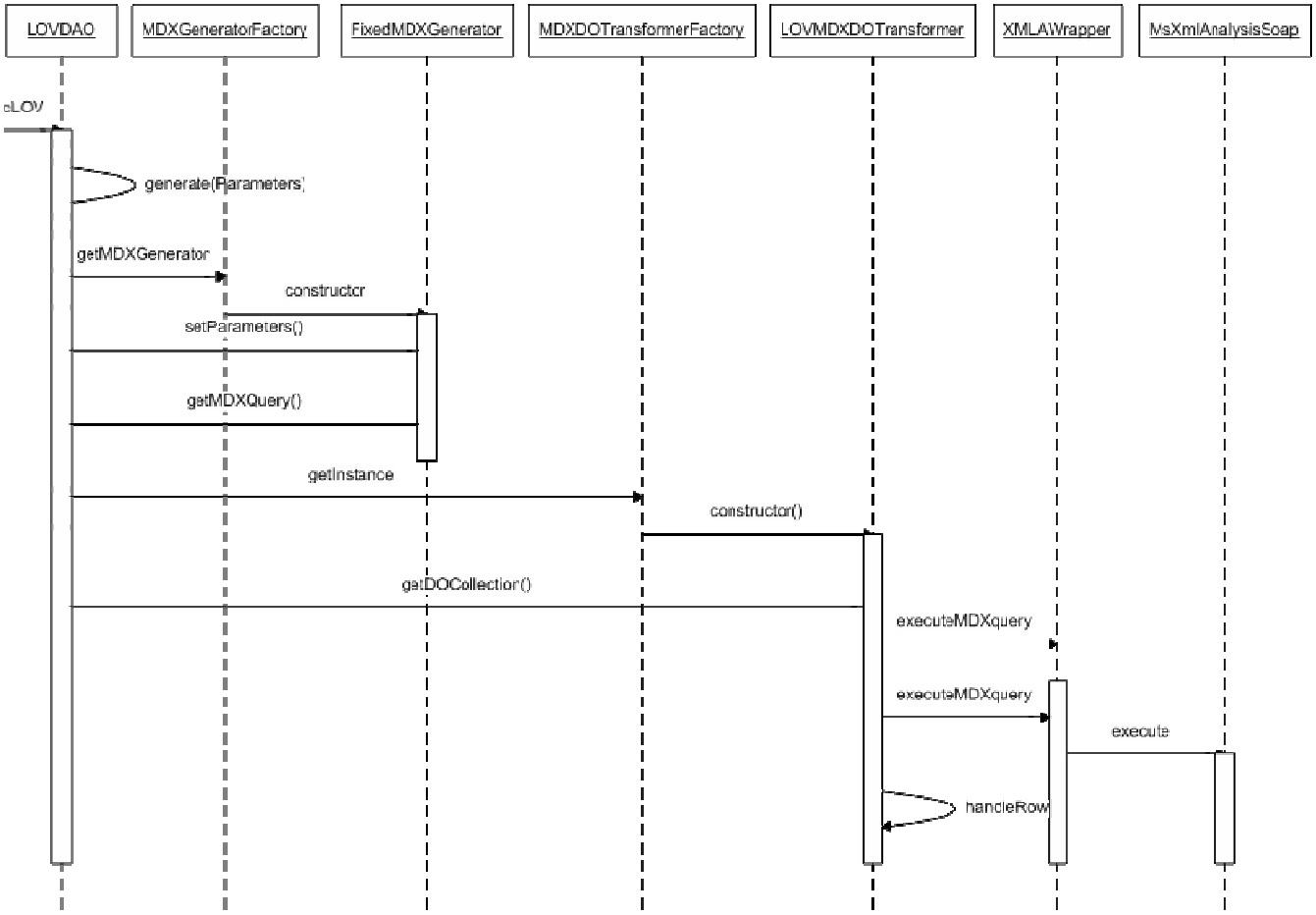

Figure 1: MDX sequence diagram

An MDX query is generated based on the query number and any additional search parameters. The correct Generator class is determined from the Generator Factory based on the query number. This means that each query (and any new query) needs to be maintained in the IMDXGenerator interface as well as in MDXGeneratorFactory->getMDXGenerator( ). The query itself is maintained in a separate .properties file since it needs to be consistent with the BEx query defined in the BW system and so may require periodic updating. There are three types of generators:

The fixed generator always generates the same query. It is used for a List of Values (LOV) service that always returns the same data for all users (e.g. list of Purchasing Groups)

The parameter generator takes a single input parameter (e.g. the material detail service takes the Material ID as input). The MDX query in the parameters files actually contains the SAP Variable name and a place holder for the value. There is a variation on the parameter generators for the multiple queries (e.g. multiple Material details).

The dynamic queries are used for the PO Header and PO Item queries. In this case, the SAP Variables section is generated completely dynamically based on the input search criteria.

Once the query is generated, it is passed to the XMLAWrapper class which actually executes the MDX query via the XML/A web service proxy. The relevant code from this class is in the method, executeMDXQuery( ):

```
/**
 * Executes the specifed MDX query by delegating the function call to
 * the corresponding WebService. The returned data is parsed into a
 * <code>BIDataSetTableModel</code>.
 *
 * @param  mdxQuery - The MDX query string.
 * @return The data model as used by the BI SDK
 * @exception DAOException - In case of an error
 */
public BIDataSetTableModel executeMDXquery(String mdxQuery) {
   // get XMLA reference (customizable with or w/o SSO)
   if (WS_LOOKUP_NAME == null )
   {
      try
      {
         Properties properties = ServiceLocator.getInstance().
                                                      getProperties();
         if (properties != null)
         {
            WS_LOOKUP_NAME =
                  properties.getProperty(PROPERTY_XMLA_WS_LOOKUP_NAME);
         }
            }
         }
   IBIConnection connection = null;
   try
   {
      try
      {
         // create initial context
         Context initctx = new InitialContext();

         // perform JNDI lookup to obtain connection factory
         IConnectionFactory connectionFactory = null;
         connectionFactory =
                  (IConnectionFactory)initctx.lookup(WS_LOOKUP_NAME);

         IConnectionSpec connectionSpec =
                              connectionFactory.getConnectionSpec();
         // establishes the IBIConnection from the ConnectionFactory.
         connection = (IBIConnection)
                              connectionFactory.getConnectionEx(null);
      }
      IBIOlap olap = connection.getOlap();
      IBIDataSet dataset = olap.execute(mdxQuery);
      BIDataSetTableModel tableModel = new BIDataSetTableModel(dataset,
                                                                  false);
      return tableModel;
   }
}
```

After the query executes, the data is returned as a `com.sap.ip.bi.sdk.dac.result.model.BIDataSetTableModel`. This object then needs to be transformed into one of our data objects. This process is specific to the query, so there are many more transformers than generators. The only transformer that can be reused is for the fixed LOV queries since

they all return data in the same format (key, description). Given this multitude of transformers, a transformer factory is used to determine the right class to use for the specified query number. So, as with the Generator factory, any new query needs to be maintained in the class MDXDOTransformerFactory->getInstance( ). The code from the transformer is relatively simple because of the BI SDK API's. For example, the Transformer for the LOV queries looks like:

```
protected BaseDO handleRow(int row)throws DAOException
   {
      try
      {
         String key = tableModel.getValueAt(row, LOV_KEY).toString();
         String desc = tableModel.getValueAt(row, LOV_DESCRIPTION).toString();

         lovDO = new LOVDO(key, desc, null);
      }
      return lovDO;
      }
}
```

Perhaps a more interesting example is from the MaterialDOTransformer:

```
protected BaseDO handleRow(int row) throws DAOException
   {
      try
      {
         materialDO = new MaterialDO();
         materialDO.setMaterialID(tableModel.getValueAt(row,
                                                        MATERIAL_ID).toString());
         materialDO.setCategory(tableModel.getValueAt(row,CATEGORY).toString());
         materialDO.setCategoryDescription(tableModel.getValueAt(row,
                                             CATEGORY_DESCRIPTION).toString());
         materialDO.setDescription(tableModel.getValueAt(row,
                                                        DESCRIPTION).toString());
         materialDO.setDivisionID(tableModel.getValueAt(row,
                                                        DIVISION_ID).toString());
         materialDO.setDivisionName(tableModel.getValueAt(row,
                                                      DIVISION_NAME).toString());
         materialDO.setGenderAgeID(tableModel.getValueAt(row,
                                                      GENDER_AGE_ID).toString());
         materialDO.setGenderAgeName(tableModel.getValueAt(row,
                                                    GENDER_AGE_NAME).toString());
         materialDO.setSegmentID(tableModel.getValueAt(row,
                                                       SEGMENT_ID).toString());
         materialDO.setSegmentName(tableModel.getValueAt(row,
                                                      SEGMENT_NAME).toString());
         materialDO.setSourceTypeID(tableModel.getValueAt(row,
                                                    SOURCE_TYPE_ID).toString());
         materialDO.setSourceTypeName(tableModel.getValueAt(row,
                                                  SOURCE_TYPE_NAME).toString());
         materialDO.setSubCategoryID(tableModel.getValueAt(row,
                                                    SUB_CATEGORY_ID).toString());
         materialDO.setSubCategoryName(tableModel.getValueAt(row,
                                                SUB_CATEGORY_NAME).toString());
         materialDO.setTypeGroupID(tableModel.getValueAt(row,
                                                      TYPE_GROUP_ID).toString());
         materialDO.setTypeGroupName(tableModel.getValueAt(row,
```

```
                                                    TYPE_GROUP_NAME).toString());
        }
        return materialDO;
    }
```

Summary

One consequence of using BW as a data source for your application is that it changes the nature of your BW system. This has consequences for uptime requirements, performance requirements, etc for the BW server. One solution is to have a second installation of BW that serves as the application data source. This would have a reduced amount of data, limited by scope to only that required for the application(s) and by time to 6 months or so.
One criticism for the use of BW as an application data source is that the data may be stale or out of date since you are still replicating data in a batch process. This was true before BW 3.5, but since then, the option exists for real-time update of BW via XI. If this is the only argument against using BW, you should think about the option of updating BW real-time. See SDN article, BI XI Integration, for more details
For more information on the BI SDK and SAP's extensions to the MDX language, see SAP Help site.
For more information on the XML/A and MDX standards see the XMLA home page and the Microsoft Article on MDX.

Author Bio

Richard Andrulis is a Development Architect for SAP Custom Development based in Newtown Square, PA. He has been employed by SAP for over 8 years beginning with SAP Labs in Palo Alto, CA as part of the Industry Solutions group for Discrete Industries. In his current position, Richard has architected, designed, and developed the first Custom Composite Application at SAP. The architecture followed the Enterprise Services Architecture philosophy and integrated Java, ABAP and .Net technologies. On a personal note, he is constantly looking for new and innovative ways to use technology to help both the developer and the end user do their jobs more effectively. Richard holds a Ph.D. in Electrical Engineering from Cornell University.

Nir Paikowsky is a system architect for SAP Suite Optimization Architecture and Technology group. He started with SAP in 2004 as a senior solution architect at SAP Labs Israel, during which he architected, designed, and developed the first Custom Composite Application at SAP. In addition to his SAP experience, Nir has over 10 years of experience in the enterprise software market in both functional and technical positions, with extensive J2EE architecture experience.

.