SOA-Centric Read/Write Methodology

TIBCO Software empowers executives, developers, and business users with Fast Data solutions that make the right data available in real time for faster answers, better decisions, and smarter action. Over the past 15 years, thousands of businesses across the globe have relied on TIBCO technology to integrate their applications and ecosystems, analyze their data, and create real-time solutions. Learn how TIBCO turns data—big or small—into differentiation at www.tibco.com.

Project Name	AS Assets Data Abstraction Best Practices
Document Location	This document is only valid on the day it was printed. The source of the document will be found in the ASAssets_DataAbstractionBestPractices folder (https://github.com/TIBCOSoftware)
Purpose	Self-paced instructional

www.tibco.com

Global Headquarters
3303 Hillview Avenue
Palo Alto, CA 94304

Tel: +1 650-846-1000
+1 800-420-8450
Fax: +1 650-846-1005

Revision History

Version	Date	Author	Comments
1.0	08/25/2010	Mike Tinius	Initial revision
1.1	08/26/2010	Mike Tinius	Revision to fix grammar and incorporate feedback from reviewers.
1.2	09/30/2010	Mike Tinius	Incorporated suggestions by David Bessemer – added overall concept.
1.3	04/05/2013	Mike Tinius	Updated layers.
2.0	08/20/2013	Mike Tinius	Updated documentation with Cisco format.
2.1	05/20/2015	Mike Tinius	Updated this white paper to Cisco format.
2.2	12/06/2017	Mike Tinius	Transitioned to Tibco for release 8.1.9
2018Q1	03/20/2018	Mike Tinius	Release 2018Q1 – no changes.

Related Documents

Name	Version
Tibco Data Abstraction Best Practices White Paper.pdf	2018Q1
Tibco Data Abstraction Best Practices.pptx	2018Q1
Tibco Data Abstraction Best Practices Technical.ppt	2018Q1
Tibco SOA-Centric Read/Write Methodology Technical Note.pdf	2018Q1

Supported Versions

Name	Version
TIBCO® Data Virtualization	7.0 or later
AS Assets Utilities open source	2018Q1 or later

Table of Contents

1 Introduction

Purpose

This technical white paper describes a methodology for performing Read/Write using Tibco Data Virtualization Server in a SOA-Centric environment. Read/Write also refers to the database atomic operations of 'Create', 'Read', 'Update' and 'Delete' which is commonly known as the acronym (CRUD).

The goal of this methodology and the framework of scripts based on this methodology are to reduce the overall effort by developers and provide a repeatable set of best practices that can be applied for any application.

The best practices outlined in this paper build upon the Read-only Data Abstraction Layer Best Practices previously defined in a separate white paper. It is SOA-Centric because all access to these services is through web services and not JDBC. Although, JDBC could be used to invoke procedures, this paper will not discuss the JDBC data access option.

Additionally, this paper will explore the use of code generation techniques to assist in reducing the work to create the services. There are six key areas that are addressed in this methodology and are described as follows:

- ***Coordinator (Save) Procedure*** – a procedure that is used to coordinate a series of 'Create', 'Update', or 'Delete' calls for a SOA-Centric service.
- ***Create/Read/Update/Delete (CRUD) Procedures*** –procedures used to implement the 'Create', 'Read', 'Update', or 'Delete' logic for a given top-level view
- ***Type Definitions*** – a procedure that defines all the Type Definitions structures for each of the views that require execution of 'Create', 'Read', 'Update', or 'Delete' operations.
- ***Retrieve Primary Key*** – a procedure used to lookup a primary key when either the primary key or alternative keys are provided.
- ***Primary Key Generation*** – A facade procedure to generate a unique key for the create procedures.
- ***Web Service Generation*** – Expose Coordinator Procedures as web services.

Benefit – Simplify the SOA BPM/ESB layer by providing a SOA-Centric data access layer for read/write. Data Virtualization reduces the work required by BPM/ESB developers for single database access.

Audience

This document is intended to provide guidance for the following users:

- **Architects** – Architects will find this paper useful to understand how Data Virtualization can be used for Create/Read/Update/Delete operations.

- **Developers** – Developers will find this paper useful for understanding how to implement Create/Read/Update/Delete operations against a single data source.

References

Product references are shown below. Any references to CIS or DV refer to the current TIBCO® Data Virtualization.

- TIBCO® Data Virtualization was formerly known as
    - Cisco Data Virtualization (DV)
    - Composite Information Server (CIS)

2 Overall Concept

This section describes the overall concept and lays out the process by which the various components will be used. The goal is to provide a repeatable read/write process for any application. The first thing that a developer would do is determine if CRUD operations were required for their application. Assuming that is the case, they would then assess their current data abstraction layers and decide which layer they will use to base the CRUD procedures off of. Typically, it would be the top-most layer which is known as the 'Mapping Layer'. However, it is not absolutely necessary to do so. Once the views are in place, the developer is now ready to utilize the methodology to implement or generate the CRUD operation procedures.

There will be a baseline procedure created for each CRUD operation. Each of these will be detailed later in this paper. From a high-level perspective, the methodology allows for one procedure for each view and each operation. For example, there are four basic operations: create, read, update, and delete. Each view would have all four procedures that implement these atomic transaction capabilities.

Another aspect of this methodology is to discover the primary key for a given view. There is a retrieve primary key procedure for each view. The objective of primary key procedures is to allow the developer the flexibility to pass in view data that provides enough information to pass as an alternate key. It then dynamically builds a query that returns a primary key.

Finally, at the top-most layer of this pyramid is what are called coordinator save procedures. The idea behind these procedures is to provide an external interface to ease the burden on SOA developers when needing the ability to hook in SOA-based CRUD services into a BPM or ESB tool. Without these services, it would be up to the SOA developer to write complicated logic for accessing discrete tables. This can be very cumbersome to do in a BPM or ESB tool. Data Virtualization eases the burden by encapsulating the logic for creates, update, and delete into the Coordinator Save procedures. The basic structure can be generated, but then it is up to the developer to complete the necessary implementation logic for their application.

The overall benefit is clear.

- The framework reduces the ground-up effort to getting started by accurately and efficiently generating baseline CRUD procedures.
- The framework provides a structure to work within that provides repeatability
- The framework provides a quick mechanism in which to generate basic web services for SOA developers to implement against thus reducing the overall start time of a project. This is important as a SOA infrastructure is predicated on having services to build from. Developing SOA Services is very much a ground up exercise. First, you have to have the Data Services before you can build the SOA Services. Application Developers can only

go so far before they need to access the SOA Services to complete their work. Like any application development effort, there are building blocks involved. Data Virtualization can help by providing efficient and quick access to SOA-based Data Services. This is very important as access to “Data” is the first building block in any application.

3 Data Abstraction Layer Review

In this section, the data service design and development efforts required to implement the Data Virtualization data abstraction reference architecture are described starting with the physical layer and moving up to the consumer. This is representative of how a typical Data Virtualization developer approaches this activity. The next section will go into more detail about the sub-layers that may be utilized during implementation.

Sub-Layer Architecture View

The sub-layer architecture view outlines four primary layers shown on the left of the diagram that contain one or two sub-layers shown on the right of the diagram. At the bottom of the diagram are the data sources or information assets within an organization. At the top are various data consumers that need access to these information assets. This section will describe the purpose of each layer.

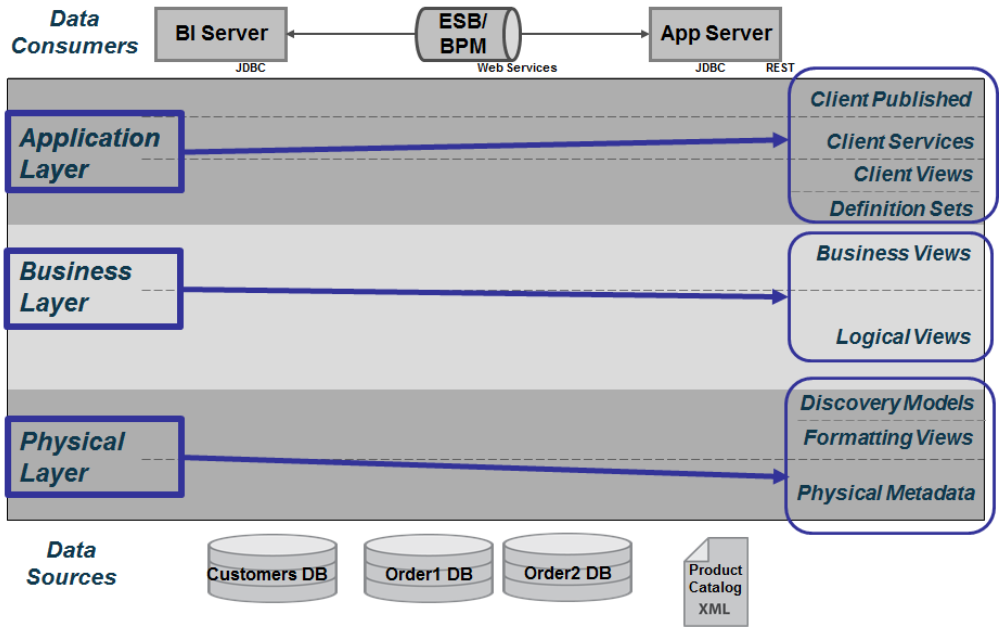

Figure: Sub-Layer Architecture View

Sub-Layer Definitions

The sub-definitions are provided as follows:

- ***Application Layer*** – this layer serves to map the Business layer into the format which the Data Consumer wants to see their data. It might mean formatting into XML for web services or creating views with different alias names that match the way the consumers are used to seeing their data.

- **Client Published** – Provides the contract with the consuming application.
- ***Client Services*** – Procedural logic for shaping results, applying parameters, custom logic and XML shaping.
- ***Client Views*** – Client Views map the Business Layer's business and logical views to the names used by the client API. Additionally, it may be necessary to pivot data for use in the UI application.
- ***Definition Sets*** – Provides a central place to manage various WSDL, schema and SQL definition sets. When working with web services and shaping of XML documents, the WSDL and schemas will be used by Client Published and Client Services resources.

- ***Business Layers*** – this layer is predicated on the idea that the business has a standard or canonical way to describing their business. In the financial industry, one often accesses information according to financial products, instruments and issuers amongst many other entities. Typically, a Data Modeler would work with business experts and data providers to define a set of "logical" or "canonical" views that represent the business. These views are reusable components that can and should be used across business lines by multiple consumers.
  - **Business Views** - Business Layer Views that implement business rules by narrowing sets of data via where clauses or aggregating data.
  - ***Logical Views*** – Business Layer Logical views predicated on the canonical business entities or concepts. These views perform regular single-source joins, federated joins and federated union of physical layer sources. It may also be necessary for the logical views to pivot columns to rows.
- ***Physical Layer*** – the physical layer provides access to physical sources and implements the first level of abstraction by mapping physical to logical names.
  - **Discovery Models** – The Data Virtualization Discovery Models provides a place to store the introspected discovery model information. Because Data Virtualization Discovery is targeted at finding relationships in the physical data, it makes sense to store the models in the Physical Layer close to where the actual physical source metadata is located.
  - ***Formatting Views*** – Formatting views perform the mapping of physical to business logical. Ultimately, physical data sources have to be mapped into this virtualization layer and it is the job of the formatting views to provide simple tasks such as name aliasing, value formatting, data type casting, derived columns and light data quality mapping. In general, these views are derived from the physical sources and perform a one-to-one mapping between the physical source attributes and their corresponding "logical/canonical" attribute

name. Naming conventions are very important and introduced with these views.

- ***Physical Metadata*** – Provides the metadata imported from the physical data sources. Consider the physical metadata as the way to onboard data sources into Data Virtualization giving developers a tangible entity create views from. Entity names and attributes are never changed in this layer. It's an as-is layer. A caveat to this is when the data source is hierarchical such as XML files or web services. In order to transition this data into the Formatting Views it will be required to translate the XML to relational using an transformation map.

Sub-Layer Naming Convention Overview

The **sub-layer folder naming conventions** overview provides a quick snapshot of the guidelines used for naming conventions of the layers. The top layer is the project name, for example: "DataAbstractionSample". There are some standard folders beginning with underscores. The remaining layers alphabetically coincide with the layers presented in the previous diagrams starting with "Application" layer, "Business" layer, and "Physical" layer as shown in the figure below:

Figure: Sub-Layers Naming Conventions

4 SOA-Centric Read/Write Methodology

Introduction

CRUD operations are the basis for being able to Read and Write to a data source. The terminology (CRUD) originates from the early days in database design where atomic transactions were executed against the database. The operations (create – a.k.a. insert, read – a.k.a. select, update and delete) provided the basis for data access. Since Data Virtualization software is based on the semantics of a SQL language, it too provides these atomic transactions for data source access. However, using atomic transactions are low level operations and don't provide any infrastructure or framework around them for things such as the following:

- Procedures for web service access
- Parameter passing using typed structures
- Consistent column naming conventions using the logical model
- Validation
- Coordination between two or more entities/views
- Error handling
- Primary key generation
- Primary key retrieval
- Custom data handling
- Testing

This paper discusses how Data Virtualization can be used to provide a framework around the atomic CRUD transactions and provide SOA-Centric data access to a database repository.

Secondly, this paper discusses the use of the Data Virtualization Repository code-generation scripts used to generate the framework which is then enhanced by Data Virtualization Developers to build out the specific portions of a given project.

Coordinator (Save) Procedure

These procedures coordinate a series of create/update/delete calls for a SOA-Centric service. The intention for the Coordinator services is to provide a framework around the overall service invocation. Even though this procedure is generated on a per view basis, it is only meant as a starting point and requires modification.

The value is this service is that it provides a template to begin customizing. Without such a template, each new service would have to be built from scratch. The problem with this is that each service would lack consistency across them and it would be time-consuming to build them. The framework seeks to apply a consistent methodology and reduce the amount of time it takes

to customize. For example, the framework provides column naming conventions consistent with the logical model described in the Best Practices documents. It provides a framework for passing record structures to the atomic CRUD operations and handling errors. It provides a consistent interface pattern for the SOA-Developers to utilize thus making the job of data access easy to use.

Create/Read/Update/Delete (CRUD) Procedures

These procedures implement the 'Create', 'Read', 'Update', and 'Delete' logic for a given top-level view. One consideration to bear in mind is that some views may contain derived columns that have no data lineage back to a physical source. Therefore, it may be necessary to create a set of views based on the logical model that provide a view implementation with no derived columns purely for performing CRUD operations. The reason this is important is that Data Virtualization will not be able to execute a CRUD operation against a view where it contains a derived column that has no lineage back to a data source.

- ***Create*** – create procedures provide a framework for executing the atomic transaction "INSERT" against a specific Data Virtualization View.
    - *Naming Conventions* – The procedure naming convention is to use create_<view_name> when generating these procedures. These procedures allow for multiple records to be inserted within a single transaction. The framework provides exception handling and logging and will return a vector of result messages for each transaction executed.
- ***Read*** – read procedures provide a framework for executing the atomic operation "SELECT" by primary key against a specific Data Virtualization View. Additionally, read procedures can be created by the developer to query by any parameter that is needed for the context of that query. For the purposes of this framework, only retrieve by primary key is discussed.
    - *Naming Conventions* – The procedure naming convention is to use get_<view_name>_ById when generating these procedures. These procedures allow for multiple records to be returned in a cursor, however because the selection criteria is by primary key then only a single record will ever be returned. The framework provides exception handling and logging
- ***Update*** – create procedures provide a framework for executing the atomic transaction "UPDATE" by primary key against a specific Data Virtualization View.
    - *Naming Conventions* – The procedure naming convention is to use update_<view_name> when generating these procedures. These procedures allow for multiple records to be updated within a single transaction. The framework provides exception handling and logging and will return a vector of result messages for each transaction executed.

- *Null Behavior* – Updates allow the developer to control the behavior of null values passed into the save procedure. For example, applications may not want to fill in every data value when updating a record. They may choose to only pass in the values they are modifying. The implication of this is tremendous. The framework needs to be able to interpret the use of null values. Either the null value is a "No Operation" or it is an actual value that needs to be updated. If the interpretation is "No Operation" or "Do Nothing" with that column, then the update statement that is generated will not include a "set" statement for that column and value. However, if the interpretation of null behavior is to force the usage of all data values null or not, then the "set" statement must be generated with all column values and explicitly set columns = null when the data values contain null. Therefore a parameter is provided for developers to define their intentions:
    - IN explicit BIT
        - 1=true, update all fields even if null
        - 0=false, only update non null fields.

- ***Delete*** – delete procedures provide a framework for executing the atomic transaction "DELETE" by primary key against a specific Data Virtualization View.
    - *Naming Conventions* – The procedure naming convention is to use delete_<view_name> when generating these procedures. These procedures allow for multiple records to be deleted within a single transaction. The framework provides exception handling and logging and will return a vector of result messages for each transaction executed.

Custom Create and Update Procedures

These procedures are akin to intercept procedures in that they provide a mechanism for the developer to apply custom logic on the data values to be created or updated.

- ***Create Custom*** – create custom procedures provide a framework for modifying the data values prior to inserting into a Data Virtualization View.
    - *Naming Conventions* – The procedure naming convention is to use custom_<view_name> when generating these procedures. These procedures allow for a single record to be modified. The framework provides exception handling and logging and will return a vector of the single row. This procedure is invoked within the multiple record loop of the create_<view_name> procedure.
- ***Update Custom*** – update custom procedures provide a framework for modifying the data values prior to updating a Data Virtualization View.

- *Naming Conventions* – The procedure naming convention is to use custom_<view_name> when generating these procedures. These procedures allow for a single record to be modified. The framework provides exception handling and logging and will return a vector of the single row. This procedure is invoked within the multiple record loop of the update_<view_name> procedure.
- *Dynamic SQL* – Because update statements are dynamically generated and executed, update custom procedures should take advantage of helper procedures for formatting the dynamic SQL according to the data type. For example, VARCHAR columns would invoke formatString() and INTEGER columns would invoke formatInteger(). The generated procedure provides a list of helper functions within the body of the procedure for developers to use.

Type Definitions

This procedure defines all the typed structures for the target views that require execution of 'Create', 'Read', 'Update', and 'Delete' (CRUD) operations. Type Definitions are created once and used in multiple procedures for Vector (record) definitions. Vectors are used as a convenient way of passing record structures between SQL script procedures. These structures can contain one or more records.

- **Definitions** – this folder contains the procedure which defines the type definitions.

According to the Best Practices, Type Definitions should be created against the layer of views that will be used to execute 'Create', 'Read', 'Update', and 'Delete' operations. Generally speaking, CRUD operations would be created against the top-level view, generally the Mapping Layer, Client Views. The Type Definitions procedure gets generated by either of these procedures, "generateCRUDOperations()" or "generateTypeDefinitions()". It should be noted that if either those procedures is executed more than once, the "Type Definition" procedure will be overwritten. It is advisable not to modify the generated procedure. If there are Type Definitions that are needed for specific application needs, they should be defined in a separate Type Definition procedure.

Retrieve Primary Key

These procedures are used to lookup a primary key when either the primary key or alternative keys are provided. The view record structure is passed into this procedure.

- ***RetrievePK*** – retrieve primary key procedures provide a framework for dynamically selecting the primary key for a specific Data Virtualization View based on data values passed into the procedure.
  - *Naming Conventions* – The procedure naming convention is to use retrievePK_<view_name> when generating these procedures. These

procedures allow for a single record to be used for the selection of the primary key. The framework provides exception handling and logging and will return a single instance of the primary key(s). There may be a composite primary key containing multiple columns.

- *Behavior* – This procedure generates dynamic SQL to retrieve the primary key. If the actual primary key column(s) contain values then it builds the SQL based on those values only. If the actual primary key column(s) are null, then it will compose a dynamic SQL statement with values from the body of the record. Therefore, the user of this procedure must insure the values in the body will compose an alternate key to uniquely select the primary key. If multiple records are returned then an exception will be thrown because a primary key could not be uniquely identified.

Primary Key Generation

This is a facade procedure to generate a unique key for the create procedures. Each project must implement the actual generation script either using packaged queries or a java procedure for the specific database being integrated.

- ***getUniqueID*** – this is a standard façade procedure which is invoked by the 'Coordinator (save) procedures' to generate a unique ID when the Create procedures are executed.
  - *Location* – This procedure is located in a standard location within the Best Practices framework. It can be found in the folder <base-project-path> /Physical/ Metadata/SequenceGenerator.
  - *Implementation* – It is up to the Developer to provide the actual implementation of key generation. The sample that is provided uses a Data Virtualization Packaged Query to implement a SQL query that invokes an Oracle Sequence. The example Package Query is shown below:
    - select cisdemo_seq_id.NEXTVAL SEQID FROM DUAL
  - *Behavior* – It is recommended that a single sequence generator be used for all Views rather than a separate sequence generator for each view. It will keep things simple and ultimately easier to maintain.

Web Service Generation

Data Virtualization Studio has the ability to generate web services any procedure. The best practice is to generate web services from the top-level coordinator procedures called 'Save' procedures. Essentially, there are two methodologies for providing web services. There are pros and cons to both approaches. Requirements may also dictate which approach is used.

- ***"Bottom-Up"*** – Web Services that are generated using the "Bottom-up" approach originate directly from the 'Coordinator (save) procedures'.
    - *Pros*
        - Very easy and quick to generate a web service. It is literally, right-mouse click and generate. The WSDL is generated automatically and can be provided to the client Developers.
        - Simple to use by the client Developers.
    - *Cons*
        - The WSDL is generated automatically and cannot be modified.
        - Each WSDL contains a single operation generated from the underlying Data Virtualization procedure.
        - It needs to be noted that all 'Coordinator (save) procedures' provide a flat interface. Consequently, the "Bottom-up" generated web services provide a flat XML interface for consumers to invoke as opposed to a hierarchical XML interface.
        - No custom logic can be applied upon execution
    - Data Virtualization *Tool* – Data Virtualization Studio is the same tool used to create the 'Coordinator (save) procedures'. It ships with the core product.
- ***"Top-Down or Contract-First"*** – this is a standard façade procedure which is invoked by the 'Coordinator (save) procedures' to generate a unique ID when the Create procedures are executed.
    - *Pros*
        - You start with a WSDL that has been defined independently of the data access layer. Therefore, it complies with the specifications and requirements the customer may for data access.
        - A WSDL can contain one or many operations but as a best practice, should be related.
        - The WSDL can contain hierarchical, nested structures. For example a customer contains zero or many orders and an order can contain one or many product detail records.
        - Custom logic can be applied in between the Request and the Response and before or after execution of the Data Virtualization procedures as long as the Java Implementation approach is used for implementing the body of the web service.
        - Uses a standard Service Component Architecture (SCA) framework to generate a composite service from the WSDL definition and associate it

with references to Data Virtualization Server (DV) 'Coordinator (save) procedures'.

- *Cons*
  - The implementation of the body of the web service requires writing custom java or XQuery code. Therefore, the skill-set is more technical than the Bottom-up generation approach.
- Data Virtualization *Tool* – Data Virtualization Designer is a separate tool used to create the 'Coordinator (save) procedures'. It ships with the core product.

Test Framework

Every procedure that is generated has a corresponding test procedure that invokes it. The value of this is enormous as it gives the developer an instant way of validating the atomic CRUD operations for a given Data Virtualization View.

- ***Test*** – this framework capability allows the Developer the flexibility to change values as needed and test the individual functionality of each generated procedure. It will display the contents of the error handling message vector upon completion of the execution of the operation. It will handle the ability to pass in multiple records for the Create, Update and Delete operations in order to test bulk operations.
  - *Naming Conventions* – The procedure naming convention is to use test_<operation>_<view_name> when generating these procedures. These procedures allow the Developer to test the interfaces and functionality of each of the CRUD operations.
    - *test_save_<view_name>*
    - *test_create_<view_name>*
    - *test_get_<view_name>_ById*
    - *test_retrievePK_<view_name>*
    - *test_update_<view_name>*
    - *test_delete_<view_name>*
  - *Location* – always located in a /test folder under the operation folder.
    - */Coordinate/test/test_save_<view_name>*
    - */Create/test/test_create_<view_name>*
    - */Read/test/test_get_<view_name>_ById*
    - */RetrievePK/test/test_retrievePK_<view_name>*
    - */Update/test/test_update_<view_name>*
    - */Delete/test/test_delete_<view_name>*

5 Generation Scripts

Introduction

Generate Coordinate, Create, Read, RetrievePK, Update, and Delete (CRUD) operations for use when performing Read/Write against a database repository. You would use the generate generateCRUDOperations() method to achieve this.

generateCRUDOperations()

Whether you have 10's, 100's or 1000's of relational tables that you want to map into a canonical format, this auto-generation utility will aid in the mapping of data abstraction layers either from other views or from physical data source tables.

This procedure is used for generating Read/Write CRUD operation procedures. In this context, CRUD includes "Create", "Read", "Retrieve Primary Key", "Update", and "Delete". Those are the base operations. In addition to those base operations, there is a coordinator which is a procedure used to coordinate the lower level CRUD procedures.

It should be noted that this procedure uses the "Target" folder in the ConfigureStartingFolders().

This procedure is meant to be executed by a Data Architect within Data Virtualization Studio.

According to the Best Practices, the following functional areas are required for performing CRUD:

1. Coordinator (save) procedures
    a. Type: Generated and Modifiable – Once generated, these procedures will not be overwritten.
    b. Objective: A template is generated for each view but is intended to be modified as needed. It only serves as a baseline to get started and reduce the effort involved in creating the 'Coordinator (save) procedures'. It Coordinates the overall invocation of one or more lower level CRUD procedures allowing the developer to invoke as many as needed, in any order that is required. It coordinates the transaction to the single database source. Provides the interface to be published externally as a web service. Values may need to be modified according to your data source. It is also recommended that if the service is published as a web service, that an external web service testing facility is utilized such as SoapUI, JMeter or some other testing facility. Once generated, these procedures will not be overwritten.
    c. Test harness
        i. Type: Generated and Modifiable – Once generated, these procedures will not be overwritten.
        ii. Provides the ability to test the service locally. Values may need to be modified according to your data source.
2. Create Procedures
    a. Type: Generated and NOT Modifiable – Regeneration will overwrite.
    b. Objective: The purpose of this procedure is to perform a 'Create' on the view.
    c. Custom Create Procedures

i. Type: Generated and Modifiable – Once generated, these procedures will not be overwritten.

ii. The purpose of this procedure provides a place to apply custom logic on the data values that will be created.

d. Test harness

i. Type: Generated and Modifiable – Once generated, these procedures will not be overwritten.

ii. Objective: Provides the ability to test the service locally. Values may need to be modified according to your data source. Once generated, these procedures will not be overwritten.

3. Read Procedures

a. Type: Generated and NOT Modifiable – Regeneration will overwrite.

b. Objective: The purpose of this procedure is to perform a 'Read' on the view by primary key.

c. Test harness

i. Type: Generated and Modifiable – Once generated, these procedures will not be overwritten.

ii. Objective: Provides the ability to test the service locally. Values may need to be modified according to your data source. Once generated, these procedures will not be overwritten.

4. Retrieve Primary Key

a. Type: Generated and NOT Modifiable – Regeneration will overwrite.

b. Objective: The purpose of this procedure is to retrieve the primary key from a view, given either the primary key or values that make up an alternate key for the view.

c. Test harness

i. Type: Generated and Modifiable – Once generated, these procedures will not be overwritten.

ii. Objective: Provides the ability to test the service locally. Values may need to be modified according to your data source. Once generated, these procedures will not be overwritten.

5. Update Procedures

a. Type: Generated and NOT Modifiable – Regeneration will overwrite.

b. Objective: The purpose of this procedure is to perform an 'Update' on the view using the primary key.

c. Update Custom Procedures

i. Type: Generated and Modifiable – Once generated, these procedures will not be overwritten.

ii. Objective: The purpose of this procedure provides a place to apply custom logic on the data values that will be updated.

d. Test harness

i. Type: Generated and Modifiable – Once generated, these procedures will not be overwritten.

ii. Objective: Provides the ability to test the service locally. Values may need to be modified according to your data source. Once generated, these procedures will not be overwritten.

6. Delete Procedures
    a. Type: Generated and NOT Modifiable – Regeneration will overwrite.
    b. Objective: Purpose of this procedure is to perform a 'Delete' on the view using the primary key.
    c. Test harness
        i. Type: Generated and Modifiable – Once generated, these procedures will not be overwritten.
        ii. Objective: Provides the ability to test the service locally. Values may need to be modified according to your data source. Once generated, these procedures will not be overwritten.

CRUD Generation Folder Structure

This section provides a snapshot into what the generated folder structure looks like. CRUD operations are generated in the following location:

<base-project-Path>/<clientServicePath>/CRUD.

An example can found in the following folder:

/shared/BestPractices/DataAbstractionSample/Application/Services/CRUD

Type definitions are generated in the following location with a default name. The Type Definition procedure name can be customized.

<base-project-Path>/constants/TypeDefinitionsGen

CRUD Folder Structure:

/**CRUD** - Default CRUD folder

/**Coordinate** - A coordinator (save) procedure is generated for every target view.

/**test** - A test harness procedure to invoke the Coordinator procedure.

/**Create** - A create procedure is generated for every target view.

/**Custom** - A custom procedure that allows the user to modify the values prior to create.

/**test** - A test harness procedure to invoke the Create procedure.

/**Definitions** - A folder containing the type definitions procedure

/**Read** - A read procedure is generated for every target view.

/**test** - A test harness procedure to invoke the Read procedure.

/**RetrievePK** - A retrieve primary key procedure is generated for every target view.

/**test** - A test harness procedure to invoke the RetrievePK procedure.

/**Update** - An update procedure is generated for every target view.

/**Custom** - A custom procedure that allows the user to modify the values prior to update.

/**test** - A test harness procedure to invoke the Update procedure.

/**Delete** - A delete procedure is generated for every target view.

/**test** - A test harness procedure to invoke the Delete procedure.

/**Utility** - A folder to place the custom utility procedures that may be needed for CRUD.

6 Conclusion

Synopsis

While Data Virtualization is often viewed for its data virtualization and federated data access capabilities, Customers should not shy away from Data Virtualization for total data access functionality including Read and Write. This paper provided a guideline for Customers who are implementing SOA-Centric Data Access patterns and need a reliable and efficient tool to provide Read and Write functionality to underlying data repositories.

Finally, this paper showed how to utilize a ‘Create’, ‘Read’, ‘Update’, and ‘Delete’ framework and demonstrates the level of sophistication that Provessional Services can bring to a project when implementing SOA-Centric Read/Write Data Access.