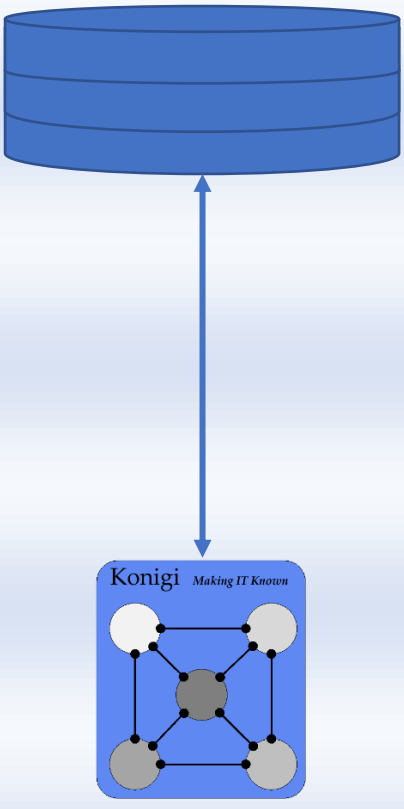
Data Access Layers

Background, Best Practices, Organizational Impact, Opposition, and Solutions



Richard L. Ladendecker
Konigi, Inc

BACKGROUND

The intent of this paper is present positions on the access of data and its dissemination, both intentionally and unintentional and to address concepts and solutions to protects its usage. After spending decades in a myriad of roles with private organizations, State and Local Governments, large and small, I decided to coalesce my experiences with regards to data and share some concepts, and best practices to ensure its access and dissemination is controlled, monitored and governed using tools and technologies available today. The driving factor of this topic is derived from observing, and collaborating with clients, peers and proponents of Data Governance and Privacy in the domains of Public Safety, Criminal Justice and from numerous engagements within industries such as manufacturing and distribution. Data was the underlying proposition for these collaborations and while the utilization of data repositories varied across government and private industries, the consistent approach to its access, control and management were virtually identical. This topic spans data privacy, governance, stewardship, control, monitoring, best-practices, transformation and integration and the reasons for the absence to many of these concepts and solutions.

At the heart of virtually every application resides data and with this comes the responsibility to manage and control its access, restrict its dissemination and promote its protection, albeit these attributes seem to be a bit lax from the prospect of being widely implemented as per news stories over the past decade. The principal of data management and control begins with the concept of architecting and developing frameworks to ensure that data can be easily accessed, can be protected from dissemination, is resilient, can be monitored and controlled, and that provides capabilities for governance. This last attribute is rapidly becoming more relevant as this concept includes logging, auditing, ensuring access rights are established and maintained and lastly, that data and its usage is accountable. This translates to who needs it, who accessed, when was it accessed, how was it accessed, why was it accessed, what happened to the data when it was accessed, who changed it and on, and on. The principal framework to support these types of elements, attributes and architectures is, for the data realm, commonly associated with what is known as Data Access Layers. As named, there are the levels of separation and segmentation of data from its repository to the consumers. A consumer is any application, co-located or remote system, a mobile device or any other technology that consumes or publishes data. Data Access Layer concepts and their implementations vary between organizations but the underlying principals and practices, when adopted, are fairly consistent and address the same organizational strategies: governance, control, management, dissemination and security.

The comprehension as to what a Data Access Layer represents to its owners is as important as implementing a DAL because without understanding what a Data Access Layer actually is, what functionality it addresses and why it is critical, it cannot be effectively designed, developed and implemented. More on this as we go forward.

The solution presented in the latter portion of this paper was conceived in late 2010 with an engagement of a project with the State of Connecticut's Judicial Branch whose IT architects, strangely enough, mandated that all data access for developed applications would be via Stored Procedures, yet not everyone in the organization followed this requirement nor was it strongly enforced. I was responsible for migrating a segment of a large legacy COBOL solution to .Net using SQL Server as the data repository. The notion of fully utilizing stored procedures proved time-consuming, frustrating and slow, yet from prior engagements with other clients, I understood that this was the most efficient method for controlling data access, supporting scalability and contributing to data management. It provided the best functionality, contributed towards efficient infrastructure performance and supported a robust framework for data dissemination while integrating their security framework. These last two benefits addressed exposing the solution's data I was migrating to other teams whose systems required data generated from my new solution in unified and consistent manner.

This project became a "Greenfield" for designing and implementing a Data Access Layer which was driven by years of prior of database activities. This project also spawned the concept for creating a set of solutions to actually generate a fully operational Data Access Layer. Suffice to say, my acumen spans a plethora of languages and data

repositories including but not limited to Dbase, Sybase, RDB, DBMS (both from DEC), Oracle, SQL Server, Access, Progress SQL, Pervasive SQL, MongoDB, NuoDB, R-Base, DB2, MySQL, and a few others. My programming proficiency includes Basic, FORTRAN, COBOL, ALGOL, LIST, C, C++, C#, VB, SNOBOL, ADA, LISP, CRONOS, APL, Macro, Assembler (multiple platforms), machine code, RPGII, LUA, PLC Ladder Logic, T-SQL, PL/SQL, Java, JavaScript, BLISS-32, ADE and several embedded technology coding languages.

Another objective for this paper is communicate the importance of implementing a DAL, the magnitude of effort to design and implement one, and the opposition to its usage within an organization. I personally have never observed one organization spanning the 70+ entities that I have collaborate with ever implement an effective Data Access Layer aside from the one I "partially created" for the State of Connecticut's Judicial Branch. In addition, not one of the client's IT Managers, Directors, CIOs or CTOs of the organizations I collaborated with and supported fully comprehended what a Data Access Layer was, its importance or how to implement one. In 2011, after performing a bit of marketing analysis and research on DAL solutions (or technologies that mimic one) and finding none that consisted of a multi-layered DAL, I initiated architecting the framework to create an automated Data Access Layer generator. The two, target databases, SQL Server and Oracle were the focus of this endeavor.

There are hundreds of white-papers, articles, manuals, documents, guides and samples of code from many talented individuals in the realm of Data Access Layers, yet I could not find one solution from any company or individual that could actually generate, build and deploy this type of technology or even come close to performing these capabilities. On the other hand, I have collaborated with numerous executives of technology solutions who claim that "their" products functioned as a Data Access Layer or "their" product eliminated the necessity to implement a DAL, just provide their solution with a full administrative level database account and they can simulate a DAL. Ok!

My stance (and that of many others in this realm) is that organizations need to "limit" exposing their data inadvertently and implement a functional DAL so when their data is exposed to these "technologies" or to developers, it is manageable, controllable, monitorable and adheres to Data Governance strategies. Another fully privileged database account is not the best-practice to "secure" a database. As one CIO I was conferring with on the topic of data governance, data controls, and Data Access Layers noted, *"I am not interested in acquiring technology and simply the providing the solution with unrestricted access to a data repository under my authority that contains highly sensitive and secure information. We cannot validate, examine or evaluate what the compiled code contained within the numerous executables and DLL's of the solution are doing to our data or ensure that the solution is exposing that data beyond our control*". The solution was a cloud-based solution similar to an ESB. Words to reflect on when you are considering acquiring technology to implement/simulate a DAL.

The solutions for a DAL are contained in this paper and I impress on data managers, CIOs, CTOs or the casual interested individuals to consider what a Data Access Layer truly represents and then examine your IT departments with an emphasis on how data is exposed and accessed in your organization.

WHAT IS A DATA ACCESS LAYER AND WHAT IS ITS PURPOSE?

There are numerous definitions for Data Access Layers, but the underlying principals go back to the 80's and 90's with the concept of "Design Patterns". A book of this name, "Design Patterns – Elements of Reusable Object-Oriented Software" written by Dr. Erich Gamma, Dr. Richard Helm, and Dr. Ralph Johnson and published in 1995 conveys pertinent concepts useful in defining a Data Access Layer (DAL). Here, the principles of "Patterns" are discussed in the realm of Software and Information Technology. Yet this is also a key concept of a DAL as the access, control, monitoring and dissemination are all repetitively implemented "Patterns". The concepts of database access are grounded by 4 underlying activities: Creating, Retrieving, Updating and Deleting data. That is, It! No matter how you work with data, you are performing one of these 4 actions. From that position, we get the concept of "CRUD", Create, Retrieve, Update and Delete, the mantra of data access.

Patterns are repetitive in that they are implemented repeatedly, have identical constructs, are architected similarly, are developed and implemented the same, and have the same traits. Translate this concept to database data access and you derive repetitive activities, CRUD. A best practice for a DAL is the adoption of Stored Procedures as the sole method to access data by and there are 4 key stored procedures (at a minimum, more later) for each database table and these are the CRUD activities; one for Create, one for Retrieve, one for Update and lastly, one for Delete. This is the initial design pattern for a DAL, four stored procedures per table supporting CRUD. A DAL has several other patterns used to access data using Indexes, and for accessing Views. Each stored procedure references a table, utilizes the table's columns, has parameters (inbound and outbound), and many other "patterns" of scripts used to access and manipulate its data. So, Patterns are a fundamental design of a DAL and necessary to ensure that DAL is efficient, consistent and well-defined. Patterns extend throughout the DAL into all three DAL layers and if Data Modeling is segmented to a layer, then Patterns will be prevalent here also.

Another concept in the "Design Patterns" publication and applicable to a DAL is the term, "Decouple" which the authors refer by "Bridge": which they note as "*Decouple an abstraction from its implementation so that the two can vary independently*." From this we propose that a DAL is decoupled from its next higher application layer, namely the Business Logic Layer. This imparts the position that a Data Access Layer can be modified without impacting the BLL or that it operates interdependently (abstraction). While the notion of decoupling seems "logical" from a conceptual point of view, it is far from accurate to believe that *complete* decoupling exists with a DAL even though many solution/data architects have endorsed this perception.

The concept of isolating data access from the BLL is fundamental to the core of a DAL just as implementing Design Patterns are, and while this level of isolation is necessary, believing that you can change a data model without impacting the BLL entirely is contrary to the notion that data drives business logic and business logic requires data. In simple terms, change an entity model element (name, address, city, state, zip, phone, etc.) by removing or renaming any of these elements and the corresponding BLL consumer which consumes them will be impacted and possibly fail; hence a DAL and a BLL are coupled whereby enhancements in one layer most often impact elements in

another, particularly when adding , removing or changing columns. How tightly coupled these layers are depends on the architecture of the DAL, the requirements of the BLL, and how the data elements are mapped, translated and exposed. Yet decoupling is fundamental to a DAL. The point is that the DAL isolates schemas, the implemented scripts to access the data, it restricts data access to well-defined and structured patterns of data via stored procedure scripts, and it prevents developers and external users from the broader functionality and features of a database, all of which can cause havoc if used maliciously. I will explain these elements further in this paper.

The underlying premise that the DAL should be "decoupled" is vital and necessary with regards to IT disciplines such as security, database performance, infrastructure, networks, data access methodologies, and isolation in order to ensure interoperability. The concept of "No-Schema Access", as mentioned, is important concept as developers truly do not require an understanding of the implemented Schemas within the database; they only need to interact with the DAL to consume and publish the data they require to support the BLL/UI, in a format that is consumable, robust and efficient. Developers should be "decoupled" from a database's design and its data access methodologies! This is not to say that developers should not participate in the database design, it mean that developers should (must) interact with the database team members in order to communicate their data consumption requirements based on information gathered by the BAs & SAs which are the application specifications used to drive the development effort. What this also translates to is that developers play the role of communicating what the application they are developing will be utilized for, what type of data is being used, how the data will be handled, reported on, and expressed. Lastly, they should communicate how they foresee gathering and managing the data from their code logic and their user's interactions, i.e. User Interfaces and the Business Logic supporting the UI (hence, Business Logic Layer). Developers should also, as a mandatory action, convey to the database teams any and all information regarding integrating any other systems within their solution. The take-away for this activity is for database team members (architects and DBAs) to formulate and implement a data model that aligns with not only the solution being presented by the developers but to also consider the other data architectures and solutions inherent within the enterprise and/or external to it. This is known as Enterprise Data Architecture which is often the realm of Enterprise Architects. Other solutions often exist within the organization and some may interact, compliment or drive the target solution's functionality. You would not want to build multiple, duplicate customer, product or inventory tables in a single or multiple database with virtually identical data elements; what would be the point? In short, data typically does not live and die by a single application especially in large organizations with numerous IT initiatives supporting almost every discipline in the organization. Data modeling and integration is a fundamental role of DBA and yet many organizations have development strategies that utilize multiple databases from disparate solutions and the development teams often are omis to disclose this caveat when participating with DBAs to develop a data model for their solution. This is an important aspect of developer to DBA collaboration and especially why data and enterprise architects are so vital to an organization; they are the front-line to understand every solution within (and external to) an organization and how they are integrated, communicate and their interdependencies.

The necessity to transform data models from conceptual business requirements into useful data schemas is inherent with almost every IT project and this requires DBAs to continuously interact with the development teams and architects to ensure naming, data conventions and dissemination are maintained consistently and that databases

and their DAL interfaces are well-defined, with changes communicated consistently and tested fully. Several large-scale projects I have collaborated on had weekly data change request/review meetings coinciding with prior change/update request deployments. As changes were solidified and agreed upon, their implementation was communicated with change release schedules. As data model changes were implemented, they were made available to limited "testing" developers typically using prototype database sandboxes to support initial evaluations of the DAL changes prior to full release to the development community for integration with their applications.

The following are examples of the hierarchy of application layers; note the instance of a Data Model Layer of the ***CodeProject*** website diagram. The intent of this layer is to abstract the native database columns into Business Naming conventions, a useful proposition opposed to performing this within the DAL Objects or in stored procedures. It would be at this level that naming conventions would be initiated and mapped to support specific application integration requirements.

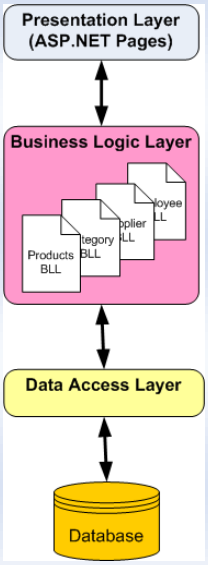


Figure 1 Microsoft Website

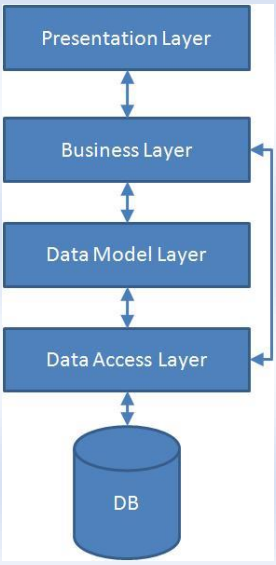


Figure 2 CodeProject Website

WHY IMPLEMENT A DATA ACCESS LAYER?

There are a plethora of important, fundamental, and best-practice concepts that drive the necessity to implement a Data Access Layer. As of the late 2010's the most noticeable and important issues focused on the dissemination of data through improper channels (theft, hacking), commonly initiated by developers and/or individuals with direct access to data repositories. Several notable data access and dissemination events making headline news were the result of bad coding practices and inadvertently failing to remove default accounts and their respective default passwords enabled for production systems. In these cases, system were "hacked" either via Web Sites or malicious code injected into operating systems whereby data was extracted and, in multiple cases, disseminated. In other reported cases, theft occurred by end-users gathering and disseminated information both manually (reports) and electronically. This scenario is almost impossible to guard against and there is no technology that can prevent physical reports from being removed from an organization and disseminated. One New York city financial entity had its garbage scavenged and it was later discovered that the

non-shredded paper contained system accounts with passwords, PII, financial accounts, and the names of database and web servers. A DAL cannot protect against this scenario.

Other reasons to implement a DAL include the necessity to implement consistent scripting and coding design patterns, to support Cyber & Security strategies, to prevent data corruption and contamination, to reduce a database's performance impact (Infrastructure, DB, Network), to accommodate Identify and Access Management & Controls, to support tiered hierarchy deployments and promotions, and to enhance Testing, Migration, Scaling, Integration, Transformation and Data Governance strategies just to name a few.

Simply stated, implementing a DAL enables an organization to sufficiently orchestrate, manage and disseminate data as a "service" that is controllable, monitored, governable and auditable. One element most IT leadership fail to consider is the viability of their data being exposed and the contingency planning to monitor and control their data to minimize or reduce the potential of its dissemination via unwarranted channels. This last point is clearly not "why to implement a DAL" but what is the impact if you do not implement a DAL and your data is disseminated or corrupted. This is likened to Business Continuity Planning and Disaster Recovery which many IT professional consider the same. The former is the planning and designing of systems and processes to sustain operations while the latter is the planning and documentation of the processes and steps to perform "when" a disaster occurs and how to restore operations. They are very different concepts. Planning and implementation of a DAL provides many benefits as will be noted next but how to handle a dissemination or corruption event becomes easier to address and manage with an effective DAL as it can provide the who, when, where and how of the event. A well architected and implemented DAL would significantly contribute towards never having to undergo a data breach in the first place.

Another key reason to implement a well-defined and architected DAL is financial: the cost of not implementing a DAL can be more expensive than implementing one. A well planned and implemented DAL can (1) significantly reduce development costs (offsetting), (2) conceal and limit data dissemination, (3) secure and limit full access through isolation and no-schema concepts, (4) facilitate controlling, monitoring, transforming, migrating and integrating systems with internal/external partners and stakeholders, (5) provide increased performance for infrastructure elements (Storage, Networks, Appliances), (6) provide a consistent point of access to data elements, (7) eliminate disparate data repositories (sandboxes, development database system), (8) minimize regression testing and the deployment of data from multiple points of development, (9) establish standardized scripts to support CRUD and custom Stored Procedure control and lastly (10) promotes entity-wide data governance initiatives. At a minimum, a well architected and instituted DAL can offset IT financial expenses in the design, development and transformation of an organization's IT Strategic Initiatives. Design and develop the architecture once and deploy universally. Developers never have to write a line of code to Select, Update, Delete or Insert data using dynamically query statements using methods like entity framework, ADO, ODBC, JDBC or any coding practice.

Lastly, a note on data dissemination and its impact. We all seem to focus on the Personally Identifiable Information (PII) of data that is "leaked" and its possible usage by malicious individuals/groups for accessing our personal and corporate financial positions. Another crucial factor is competitive advantage to improper data dissemination. What happens if your competitor gets your "key" data? This

could be in the form of contractual information, patent or copyright data, legal information that constitutes criminal or civil proceedings. This form of data dissemination is more prevalent as companies most often do not realize their data is being hijacked and/or sold to competitors. To illustrate the breadth of this situation, consider the number of contractual developers working on your applications, either on-shore or off-shore and who have these people worked for in the past. Did any of them work for your competitors? How do you know they are not disseminating your data consistently? In the Criminal Justice and Public Safety realm, this take an ever more sensitive aspect of control. Considering that Law Enforcement and Criminal Justice Systems have very sensitive data regarding events, places, people and allegations/charges, these systems should be "locked down" to the highest levels with a DAL and Appliance/Surveillance monitoring to ensure dissemination does not occur at any point. The FBI has a CJIS (Criminal Justice Information Systems) division which is responsible for working with each state at a "single point of contact" to ensure FBI Criminal justice related information is not disseminated or mis-used within states by Law Enforcement and Criminal Justice agencies. Yet every Law Enforcement RMS system I have worked with (8 of them) failed to implement a secure Data Access Layer control methodology and is often not restricted in that data is readily queried by their applications dynamically. I have used multiple RMS system where data access is purely "dynamic" and the SQL, Oracle or Pervasive SQL database is "open". Many of these COTS system vendors have a long way to go to isolate and lock-down their data repositories for their applications and this include Public Safety, Criminal Justice, Corrections and Court Case Management Systems. The same is true for some of the large ERP solutions available today. Lastly, and perhaps just as vital is National Defense and the agencies that encompass this realm and their data repositories. Data Leaks in the news have been prolific over the past 2 decades and the ones in the news are not the only ones that are occurring. What about the ones we do not hear about and what initiatives are being taken to isolate and secure sensitive National Defense and Intelligence Data? A DAL is the first and foremost point of control, monitoring, standardization of data access, and governance yet very few agencies actually implement a functional DAL.

WHAT ARE THE BEST-PRACTICES FOR IMPLEMENTING A DATA ACCESS LAYER?

Implementing a DAL can be varied and encompass many strategies and frameworks but suffice for this document, let us focus on the usage of a DAL with Microsoft's SQL Server and Oracle Database, two of the most widely used Relational Databases. Both of these support relatively the same feature sets including Tables, Keys, Indexes, Foreign Key Constraints, Stored Procedures, Triggers, Functions and Views. Their underlying capabilities and engines separate them with regards to scalability, fault tolerance, implementation methods, scripting syntax and their hierarchy and characteristics for how they define schemas.

There are often 3 layers that are often used to implement a DAL; (1) Database Data Access, (2) DAL Objects, and (3) Web Services. Each layer provides a successive level of isolation, yet each layer reduces the ability to expose a data table column's characteristic. While this is not critical, it is nonetheless important as the underlying table's column characteristics must be communicated with the development team to ensure consistency and compliance to limitations defined by the DBA team and/or other solutions implemented in the organization or due to external data sources. More on this later.

The 3-layer concept is not fixed as there are variations that support Data Modeling, and Data Transformation as additionally layers depending in the requirements of the organization and the complexity of the solution. Suffice to say, a DAL can be reduced to a single layer, that of the Database Data Access layer utilizing Stored Procedures. An IT group may limit its DAL design to provide this single layer of isolation which will contribute significantly towards limiting data dissemination and improve data control. The optimal goal is to transform a DAL into providing "Data as a Service" (DaaS) which is the most efficient and modern objective of many large organizations and Cloud providers such as Microsoft, Amazon and Oracle. Data as a Service, while the epitome of data access is not without strings attached. Getting to this level is complex and cumbersome but the end-goal is worth the effort and expense. One consideration for DaaS is that data "must" be "locked" so that its lowest level of access and dissemination protects it from unwarranted access, particularly from dynamic queries. By this, I am referring to utilizing "only" Stored Procedures to access/modify database data. The notion of solely utilizing Stored Procedures is in essence the "Holy Grail" of data access control with respect to relational databases yet it is the least implemented functionality in the relational database realm. If you start with this premise, Stored Procedure only for data access and control, then your DAL will have resilience and your database will be virtually protected from being disseminate maliciously. Also, remove everyone's database access privileges (except DBA's) to connect, query or access your databases dynamically to support this proposition. On the other side, COTS product providers profess their solutions transform a customer's data to expose it as "Data as a Service", and while this statement is correct, the data is not locked down, and is accessible by dynamic queries using accounts with elevated privileges which can be utilized to query and modify the data (and potentially the schemas). If the ESB, API management tool or other technology is not managed by the database team, then you are still exposing your data to possible dissemination and corruption by individuals who are not fully responsible for data governance and stewardship. Many organizations I have collaborated with leave the ownership of such tools to the development teams. Think about this situation for a moment, you have DBA's yet give the development team access to a tool that can do anything to a database! Further, what is the IT position on other tools that can read/write/update and delete database records with dynamic queries. This could be in the scope of ad-hoc reporting tools, ETL tools, integration tools, and others. What is your IT policy on the usage of these technologies with regards to data governance, data security, database performance and dissemination of data?

LAYER 1: DATABASE DATA ACCESS LAYER

The first layer, Database Data Access, is accomplished most often via architecting, developing and implementing consistent, templated Stored Procedures. From a utopian Data Access point of view, an efficient DAL would fully implement Stored Procedures as the sole and primary method of data access. There are multiple underpinnings to support implementing Stored Procedures, each of which has its unique characteristics and benefits. The most important one is the standardization of data access using a scripting language that can be templated to form concise, consistent and universal routines to access and control data. Stored Procedures should be designed to query data using only Keys, Indexes, and Foreign Keys as this prevents table scans and keeps the query focused to a singular table which is atomic; it stands by itself. This statement targets a 1:1 point of access for each table using the CRUD abbreviation (or by RUD in the case of Indexes). Table Joins and multi-table access will be covered later as these are specific to "Custom/Customer-Specific Stored Procedures" opposed to pure CRUD Stored Procedures. Another critical yet relevant point for implementing Stored Procedures

is that they can be designed to prevent “SQL Injection” performed by executing dynamic statements from other SQL Stored Procedures, or passed as commands from commonly used data access frameworks such as ADO, ODBC, or JDBC which can results in data being exposed, or corrupted. SQL Injection is one of the most prevalent forms of data contamination and inadvertent database or system access. A dynamic statement can return information from the execution of an O/S command, execute another program, shut down a firewall, send an SMTP message or even just return the schema of a database. Dynamic scripting is extremely powerful and can allow external control and access to not only a database but also the OS of the host system. Dynamic SQL injection can potentially execute OS commands, as an administrator, depending on the configuration and installation of the database. SQL Injection is a method widely used by hackers (and naïve developers) to access and expose data and information using the database as a remote engine.

With most IT departments, developers are able to access data any way they what, typically implementing the easiest, most widely used methods, either mandated by their organization or by which they are most familiar with. The problem with this: Developers are not DBA’s and DBA’s are not Developers. What ends up occurring is the database query statements created and embedded by the developers within their code is dynamic and these query statements typically include non-indexed and/or non-key columns which results in database table scans, or the developers do not consider database data access rules such as those implemented by Foreign Key Constraints. This one point is sufficient enough to warrant standardizing data access methodologies using stored procedures as dynamic data access is prone to incorrect syntax, improper usage and often, can be used to “back-door” systems. Developers can easily create TCP/IP endpoint listeners within their code allowing for remote access to their applications or code their applications to reach out to an internet endpoint and wait for commands. This can be used to access other internal systems via one application within an organization. This process is easy to implement and can funnel vast amounts of data seamlessly and without proper monitoring and controls, and unbeknownst to anyone, just like an Avatar. Code concealed within obfuscated and then compiled and references as a DLL can easily be hidden from scanning software and opens external channels via schedules or external triggers. Combined with implementing dynamic queries and you have a situation where a single application can pull all the data from 1-n databases depending on access controls and privileges. The data can be easily encrypted, compressed and sent in short packets to evade detection.

To further promote the implementation of stored procedures, they have the ability to be designed to implement a wide variety of control and monitoring actions such as error handling, transaction control, data access logging, authentication, token control, IAM, and they can promote the adherence to naming conventions. This last point is an important element as this aspect helps the DAL maintain its isolation with the BLL and the User Interface (UI) by helping to support consistent naming of columns necessary to drive the BLL. Changes in the data model need to support consistent usage by the BLL. If naming of data columns changes, the consumption of the data at the BLL level will have to change to accommodate the DAL change. If the DAL maps the new column name to the current BLL consuming identifier, then the mapping helps promote BLL/DAL isolation, a fundamental point of a DAL or a DAL Model Layer.

The next best-practice element that is associated with Stored Procedures is the implementation of a single account with “Execute Only” as its sole privilege. This forces all access of a database to be via execution of Stored Procedures. Through the elimination of all

dynamic query statements (coded query statements using ADO, ODBC, JDBC methods) and forcing execution of Stored Procedures virtually eliminates exposing a database schema to anyone outside the DBA team. SQL injection is also virtually eliminated (not speaking for custom/customer developed stored procedures that implement "exec" statements against dynamic statements) and the inadvertent query with a non-indexed or non-key column is also eliminated preventing unnecessary table scans. Instead of providing multiple DB or user accounts to developers to access a database directly (and the inherent management, resetting, cancelling, changing, etc.) a single account is created with a sole privilege that limits control to "Execute" pre-written Stored procedures. This is known a Locking-out a Database. At this point, we could proceed with providing developers with Execute-Only account information allowing them to code their applications with their existing access methods. The only difference would be the elimination of dynamic queries as only Stored Procedures could be called with the account privileges provided. In this scenario, all of the stored procedure definitions would need to be documented and provided to the development teams to support their coding activities. This includes Stored Procedure names, parameters, data types, scale, size, and parameter direction. Developing the documentation and supporting the continual changes can be a lengthy process but necessary to aid the developer's implementation of these stored procedures.

LAYER 2: DAL OBJECTS

If we follow a well-defined DAL architecture, the next logical step would be the execution of the Stored Procedures via code which would be provided to the developers. This methodology eliminates the prior concern of fully documenting the stored procedures for the developers as the compiled code will present the parameters and their datatypes seamlessly. There is still documentation required for this level but not the magnitude of just exposing the Stored Procedures to the development team. There are nuances to this as scale, size and direction are typically not parameters of a function/sub-routine, only the parameter name which can impact coding efforts. To clarify, if a data column is 50 characters, the function would expose a parameter of a string in the DAL Objects, regardless of size. The database would truncate any characters beyond the 50 defined at the column level. The same issue is relevant for decimal scale, binary object sizes, etc. Any database column limitation would not be easily exposed with coded functions/sub-routines. At a minimum, the column constraints for the exposed parameters must be documented and communicated to the development teams to ensure interoperability with the DAL. Another issue with function/sub-routines are passing back multiple parameters such as error fields, or additional returned values. In this case, well-defined classes are utilized as these become the returned objects from a function. Best practices here are to either return JSON, XML, Datatables or Datasets (the last 2 can be serialized as XML or JSON if required).

DAL Objects are the link between the database and the developer's Business Logic Layer (BLL) and often includes exposing each Stored Procedure as a Function or Subroutine with responses (for functions) with a myriad of options (strings, classes, objects, data types, Datatables, Datasets, xml, json, etc.). The function of the DAL Object is to bridge the gap between the Stored Procedures and the developer's code, and these functions clearly denote the parameters required by the Stored Procedures and any retuned elements emitted by the execution of the Stored Procedure. The DAL Objects are tightly coupled to the Stored Procedures, name by name, parameter by parameter and allow the developers to concentrate on consuming data without having to know what stored procedures

are available and the parameters and their data types to consume/publish data. The DAL Objects often have the connection elements provided via XML or text format which accommodates changes necessary for promotion between the SDLC hierarchy (DEV, TEST, UAT, PROD) and limits exposing unnecessary elements of the DAL Objects to the developers. DAL Objects are most often available as a code library (DLL), or as raw code in the form of modules or classes which can be included and compiled into the BLL solution. There are advantages of raw code and for DLL's, but this decision is usually left to the developers, Cyber and Security groups and Enterprise/Solution Architects of the organization. Raw code affords the development team flexibility on executing and controlling database calls. There are advantages for implementing transaction control at the DAL Object level opposed to the Stored Procedure level. Oracle and SQL differ on their implementation of Transaction control so extending this to the DAL objects further couples the DAL Objects to the Database Data Access layer. As an alternative, DLL's are the most efficient method to "secure" the code logic preventing unwarranted or malicious attempts to gain access to the Database Access Layer (stored procedures) and the inherent code used to interact with the database as this is provided as compiled code; hence non-readable. This limits the ability to expose to developers and other parties of interest, to identify the names of the stored procedures being executed and limiting access to the parameters directly passed between this intermediate layer and the database's stored procedures. With DLLs, developers are usually provided with the Account ID and password either within configuration files (xml), as encrypted strings, or via plain text. Within the embedded DLL could be error handling routines to write to the OS's event log or to a Syslog on a remote system. It may also send SMTP error messages to a DEV/OPS team or write ITIL records to a management and orchestration platform. Again, there are use-cases for both DLL's and raw code, but the intent is still the same; provide a consistent, well-define method to execute the Stored Procedures.

I personally follow the precept that "less is more" and utilize DLLs more often and this is often the scenario when this DAL sub-layer is the highest level an IT group decides to implement for a DAL.

As a closing note for this section, DAL Objects are often written in the same language/framework that the developers are utilizing, not that this is a limiting factor as almost any coding standard/language can be utilized as long as the code or the DLL can be accessed/referenced by the developer's code. If source code is provided instead of DLL's, they should be in the same language as the developers are using otherwise multiple compilers, potentially multiple IDE's will be needed to compile and integrate DAL Objects with one another; this is not an appealing proposition from a development standpoint as this is very inefficient and cumbersome to diagnose, support and manage.

LAYER 3: WEB SERVICES

The final DAL sub-layer we review and the most overlooked and underutilized is exposing the DAL Objects to developers, consumers and partners via Web Services. Implementing Web Services as the final stage of a DAL establishes an architecture whereby DAL Objects are made available to developers using managed and controlled transports and protocols, most often HTTP/HTTPS. Over the past two decades, many IT departments have realized that web services can promote faster, more efficient and unified development methods of applications by implementing a DAL using Web Services as the final Data Access layer. Again, this architecture eliminates the ability

for developers to access data “any way they want to”. It is the next logical step beyond directly integrating DAL Objects via code or code library and even eliminates the direct execution of stored procedures. Implementing a DAL that exposes data via web services using SOAP or REST provides a uniform, efficient, controlled and monitorable framework that would be difficult to match using conventional ADO, ODBC, JDBC approaches and the security access issues that are alleviated due to related dynamic queries.

Consider an Enterprise that is either small, medium or large in scale with their distinct data demands from developer’s applications, other divisions, partners, supply chain stakeholders, both internally and externally. By changing their data access architecture to utilize web services, IT departments would be able to leverage security appliances from CISCO, F5, Barracuda, and Alcatel for network and traffic control of their data. They could further enhance their data domains by complementing the flow of information with technologies such as Enterprise Service Bus and Message Queuing from SoftwareAG, Microsoft, IBM, and MuleSoft or by implementing API management with tools such as Perforce’s Akana. These technologies contribute towards a “Best of the Best” data governance and control architecture and can provide diverse and prolific sets of capabilities, all based on a fully architected and implemented DAL. The benefits realized by utilizing these messaging control technologies is that data can be directed, controlled, monitored, managed, coalesced, and processed throughout an enterprise seamlessly and efficiently. Implementing these technologies can transform an organization’s IT data architecture to provide data seamlessly to its consumers using powerful, robust frameworks that provide the best-of-breed for an Organization’s Enterprise IT requirements. At this point, we transform Data Access into “**Data as a Service**”.

WHAT IS THE ORGANIZATIONAL IMPACT OF A DATA ACCESS LAYER?

Implementing the “Best Practices” for a DAL can promote significant financial and operational benefits if taken to their fullest path and architected and developed efficiently and consistently. The cost to implement, manage and control a well-defined DAL has the potential to significantly offset other IT costs and improve the overall efficiency of an organization’s IT department. This translates into shorter development cycles, faster data model to code implementations, significantly greater management and control of data, and providing virtually any conceivable manner to consume, publish, control, trigger, transform and audit data and its utilization within an organization.

A fully architected and implemented DAL can provide a significant, positive impact on an organization across multiple IT disciplines including Cyber, Security, Infrastructure, DEV/OPS, Development resources, Finance/Budgets, Transformation, Migration, Integration, Research and Development and M&A activity.

A notable aspect of a data dissemination as of the 2010’s, is within the Cyber-Security realm. Over the past twenty years, we have heard and read about hundreds of organizations whose data was disseminated, hacked, and corrupted. Why did this happen? How did this Happen? How could this have Happened? These are just a few of the repetitive questions that society, governments, and the principles of these organizations ask, and it all returns to the fundamental issues of “Data Access”.

Throughout my career, I have worked with more than 70 entities ranging from small businesses to fortune 100 conglomerates, to State and Local governments on a myriad of IT projects and every one of these organization’s IT departments allowed their developers

to directly access, copy, manipulate and disseminate their data at various stages of their projects. This includes companies like GE, Alcatel, Tyco Labs, the State of Connecticut, and Santa Clara County. In one engagement the project retained consultants in India, California, Texas and Georgia all accessing data with Personally Identifiable Information at the database level. In every case, developers were provided “sandboxes” of databases (a copy of test or production) with “live” data that they could use for development, testing and prototyping.

While it is necessary to have data to design, develop and test code, providing the entire database to multiple developers is one factor as to why data is going where it should not. One disgruntled employee/contractor and your data is in the wind. If one of your developers/contractors needs money or is coerced into giving up your data and they have access to it, consider it gone. Data is priceless to the right people and by providing open, unmonitored access of data to your developers you are actually contributing towards exposing it without controls or restrictions. Contracts and agreements not to divulge, discuss or disseminate are “virtually” useless and the larger the organization, the easier it is to hide these exposures. Just consider the impact of Edward Snowden or Bradley/Chelsea Manning. Data expose has occurred across all industries, in government agencies, in the defense departments of most countries, in financial institutions, in insurance and health care organization, in the intelligence communities and too many more to list. If you can code an application, you can disseminate the data… Plain and Simple. The point is “how to restrict and control this”?

The cost of lawsuits, civil and criminal proceedings against an organization and/or one of its principles due to data dissemination can cause havoc on an organization. This not only impacts the principles but also its employees, partners (supply chain), shareholders/stakeholders, boards of directors and as observed with some of these large financial entities, the general public. Just use Google to search for “Data Breach” or “Data Leak” and see the number of records exposed and the organization whose data was exposed… Banks, Military, Finance, Intelligence, anyone and everyone is a risk of losing data because of the failure to properly implement, manage, control, and monitor a Data Access Layer. The financial risk is the other element to consider for not properly securing data as many of these organizations have been levied significant fines by country and state governments, through class action civil lawsuits. Uber was fined 150M, Equifax 575M in the US and 1M in the UK. Others include Marriott International with 124M, Yahoo 85M, Tesco Bank 21M, and British Airways 230M. The cost to implement and secure their data using a well architected and managed DAL would have contributed significantly toward protecting and controlling these organization’s data. While a DAL cannot guarantee complete protection, it can significantly contribute towards restricting its dissemination and the resulting malicious results. The other aspect of why this data was accessed is related to poor coding practices, inadequate database and network security issues, poor developer environment management and the intentional exposure of systems by developers (back-doors).

Other benefits of implementing a DAL can be realized by several IT disciplines and financially. First, implementing a well-defined DAL will enable developers to gain access to data in an orderly and succinct manner. Since a DAL isolates a developer from directly accessing the data repository, they should not have to design, develop, and maintain the layer or code necessary to implement the basic CRUD (Create, Retrieve, Update and Delete) methods. Developers should be able to actively access the data elements they need to drive their business logic and the corresponding User Interface (UI). This amount to a considerable amount of resources to support design,

development, testing and maintenance of code. With medium to large projects, this can constitute a considerable amount of time, resources, budgetary cost and delays with a project.

Security is the next critical element since a DAL disassociates the developer from having direct database access hence they do not need an account to access the data, they do not have a copy of the database and the management of this element alone contributes to effort (design, development, testing, deployment, maintenance) for the projects resources. Infrastructure also sees significant gains as the DAL would implement best-practices for database data access by implementing Stored Procedures and scripts that would comply with rules for accessing and controlling data (predicated searches, updates, inserts, deletes) with only key/index columns. Since there would no islands of "sandboxes", data would be consistent, licensing for multiple databases for developers would be reduced, copying, updating and disseminating data would be eliminated (developers need to keep their sandbox database aligned with each other and a baseline database).

Performance, next on our list, is another under evaluated area but nonetheless, critical to an organization. With the implementation of disseminating database via sandboxes, every developer may have multiple databases running on their development workstations and/or multiple database servers, either physical and/or virtual. Both scenarios may require licensing, but the latter requires effective storage platforms, network access (multiple ports, IP addresses, etc.), security access (multiple accounts), equipment/virtualization costs and management and synchronization the data. Here was have duplicate databases that are not synchronized which requires resources to maintain their data elements, islands of data. Adding costs for both human and technology resources can magnify if every developer has their own repository or multiple repositories are created for development teams for each application. Keeping every repository synchronized, backed-up, anti-virus protection updated, accounts setup and managed, etc., etc., etc.… The underlying costs affects not only performance of the infrastructure but also budgetary performance as well.

As noted previously, Programmers are not DBA's and DBA's are not Programmers. With that statement we need to consider "how" a developer accesses and manages data. There are fundamental database rules that impact accessing and controlling data and these focus on database concepts such as Indexes, Keys, and Foreign Key Constraints. Other elements include triggers, views, functions, and stored procedures. While we can dive further into a databases operational controls, these 7 are the most important at this stage. I note these based on my observation of coding practices across every entity I have engaged with across my career and through collaboration with peers. The prominent terms are the first 2, Keys and Indexes, and these are the golden attributes that every DBA is instilled with in their training. The design, implementation and management of Keys and Indexes within a database and their usage in Stored Procedures, Functions, Views and Foreign Key Constraints will determine the efficiency and performance of the database and its corresponding support infrastructure. The leading sentence clearly notes the delineation of developers to DBA's hence the importance of implementing a well architected and implemented DAL; in essence, removing the onus of properly controlling and accessing data from the developers as a strategy and as an architecture.

WHY ARE DATA ACCESS LAYERS NOT COMMONLY IMPLEMENTED? WHAT ARE SOME OF THE OBSTACLES?

As with why it is necessary to implement a DAL, there are as many reasons as to why a DAL is "not" implemented or at a minimum, are not fully implemented to provide the maximum level of benefit and contribution to an organization. Several of the most significant reasons that DAL's are not implemented are (1) again, Developers are not DBA's and DBA's are not Developers (who owns this layer) (2) There are a large number of frameworks and technologies to "access" data which drives most IT departments to leave data access up to the developer's discretion. (3), DBA's are too busy to fully design, develop and implement the scripts and elements necessary to implement the framework to support a DAL, (4) a DAL can be very large and daunting to architect, design, develop and implement manually, and last (5) there are few, if any, tools to support building a complete DAL or available as a COTS solution.

One common misconception which IT departments fall back upon is that tools are being utilized that can "simulate" a DAL using technologies such as Enterprise Service Bus (ESB), Queuing or Data Access Providers. This is apparent with the marketing by these product's vendors and their distribution partners who cite that they can "provide your data to you any way you want it". This seems to be a universal and "fantastic" marketing statement but be wary; these technologies (1) require "Full – Administrative" access to your data repository and (2) they do not consistently follow database best-practice methods for accessing tables using Keys, Indexes and Foreign Key constraints. They all implement "dynamic" queries against your database by whatever field is coded in the queries developed by your implementation team without regard to these Keys and Indexes. In every instance I have collaborated with client who implemented these tools, the gave the "ownership" to the development teams, not the DBAs. Again, Developers are not DBAs and DBAs are not Developers. Since these solutions do not address best practices to access data, they can significantly impact the overall performance of your database just as a programmer does with poorly written, dynamic data access statements in their code. These technologies also present a Cyber-Security risk as they constitute a point of ingress/egress to your data from an outside technology. The implementation of an ESB or Queuing Technology needs to be initiated with the DBA's involvement as improper/inadequate queries developed with these technologies has the same effect as a developer executing a dynamic query within their code. I have been told by multiple marketing and technology executives of these products that their solution can "access a database seamlessly" and there is no need to implement a DAL. Yet when improperly used, these technologies can severely impact a data repository's performance to the point of making it unusable.

On the other hand, the most versatile ESB and Queuing technologies can seamlessly integrate with a well-designed DAL at any of the DAL intermediate layers namely the Database Data Access Layer, the Data Object Layer or the Web Service endpoints. These types of integration capability provide the most robust forms of flexibility to manage your data access requirements, promote concise and broad data governance initiatives and contribute overall towards any data management strategy. IT departments retain full control of their data and expose only those Endpoints, Data Objects or Stored Procedures that are required by the BLL or peer consumers. The key is to start with a viable DAL. These solutions have to potential to provide incredible capabilities that can transform an organization's IT department into highly efficient and effective one.

Manually designing, developing, managing and coordinating the usage of a DAL is time consuming and labor intensive and changes to the underlying data models impacts stored procedures, functions, and views and require exhaustive corresponding modifications to

all 3 DAL layers (if they are all implemented) with the related regression testing of each layer's elements to ensure proper exposure of data to the developers is succinct and accurate. Considering that most medium to large organizations have a Developer to DBA ratio between 20:1 to 40:1, it is no wonder that DAL's are not fully implemented and data access at the discretion of developers or acquired technologies that "simulate" a DAL. The other issue is "who" owns this DAL development realm? Again, back to Developers are not DBAs and DBAs are not Developers, yet this environment needs to include both scripting language and high-level language development to implement a DAL. Who designs, builds, deploys and manages a DAL? The development team. Certainly not! The DBAs? They are not developers! Only one organization I engaged with actually had a data development team whose sole responsibility was to manage and control data access yet they did not implement an operational DAL, only provided access to data, upon request on a case-by-case basis and most often with hard-coded (and dynamic) routines to publish and consume data. The output were most often DLLs.

The magnitude of code required to implement a DAL is another consideration that impacts the implementation of a DAL in an organization. For each database table, a set of stored procedures is required to retrieve, update, create and delete data. This is commonly abbreviated as "CRUD" which represents "Create", Retrieve", "Update" and "Delete". Database nomenclature is actually Select, Update, Insert and Delete but it appears that CRUD was adopted as the letters "S", "U", "I", and "D" do not seem to be arrangeable to be anything meaningful; not that CRUD makes sense but that definition has become the de-facto term for the group of methods used to access data.

To reiterate, the key element for a DAL are the implementation of Stored Procedures which are prepared scripts that execute commands based on the database's scripting language; namely T-SQL for SQL Server, PL/SQL for Oracle, etc. A stored procedure allows access to a Datatables via the database scripting language. A well-defined stored procedure utilizes best-practices for selecting (the "R" as Retrieve in CRUD), Updating, Deleting and Inserting (the "C" as Create" in CRUD) based on the above noted Keys and Indexes. Searches, Updates, and Deletes must adhere to keys and indexes otherwise improper access of data using non-key or non-index columns will results in table scans which puts an undue burden on the database and its supporting infrastructure. While small tables do not readily impact overall performance, large tables (millions of records) can significantly impact the database's response with excessive table scans due to queries of data using non-predicated columns (elements of the query include columns that are not part of an index or a key).

In simple terms, if you have a datatable with a large number of columns and a large number of records and you select, update or delete data using columns in the query that are not part of Key or an Index (non-predicated) , then the database engine will select every row and build a "temporary table" in order to select the rows/columns contained in the query. This will significantly impact the overall performance of the database engine as it requires extensive memory and CPU resources, and this impacts storage by excessively increasing disk IOPS in the form of read cycles. Infrastructure elements become overburdened when applications make hundreds of queries that span of hundreds of tables and large numbers of records. Once saturation is reach, performance will degrade to the point of the application being unusable. This can be devastating to both consumers and IT DEV/Ops as most often, the

infrastructure supports more than one application, all of which are impacted by this issue. The cliché of get add more CPUs, add more disks or add faster disks can only offset this situation so far.

Now we approach the “Math” segment of this paper and the most illustrative reason DALs are not fully implemented. First off, we commonly require 4 stored procedures for each table representing which is known as “CRUD”, as defined previously. Next, we have the Indexes, and these are used to create internal database tables that enable the database engine to organize a table’s data for rapid access. These tables have pointers to the actual data records so when a query is made against the data table, the index table points to the relevant records using internally generated identifiers. This concept combined with unique “Keys” give the database its power and scalability to manage data effetely without having to result to old-fashion file-access methods using ordered-lists, sort-lists, etc. To utilize these indexes we need, at a minimum, one stored procedure for each table CRUD method. A table can have multiple indexes consisting of 1-n table column, either singly or a group of columns and for each index, we need at least one stored procedure to execute it. The “at least” is important since, some developers utilize indexes to update and delete data just as they do with selecting data. If indexes will be used to update and delete data, then, optionally, we need 1 more stored procedure to update records for each index and 1 more stored procedure to delete records by.

As a basic mathematical equation of work effort and magnitude; consider this formula.

- 4 x (CRUD)/Table + 3 x (RUD)/Index
    - This translates to 4 Stored Procedures for every table, (Create, Retrieve, Update and Delete) + one Stored Procedure for each Retrieve by Index and if you implement Updating and Deleting by Index, then +1 for each Update by index and +1 for Delete by index.
    - 1 Table with 1 Index (with Select, Update and Delete by Index) = 7 Stored Procedures

Tables routinely have multiple indexes depending on the table requirements (normalized/denormalized). Denormalization can increase the number of indexes for a table significantly but for simplicity, we will consider this as a normalized table (minimal columns and few indexes). For this example, we will have multiple indexes for each table.

Let us propose that a data table consists of 1 to 3 indexes, which translates to a DAL having 7-13 Stored Procedures for each table. Let us also consider that a small database supporting an application may have 5-20 tables, a medium size database would have 20-40 tables and a large database would have 40-200+ tables. Typically, ERP/MRP solutions which I have worked with contained 350+ tables and several Law Enforcement Record Management Systems (RMS) consisted of over 200 tables. Again, well defined databases have numerous indexes for each table so as a min/max, let us use 3 indexes as the minimum number of indexes and 7 indexes as the maximum number. We will consider small, medium and large databases in this example.

The following table illustrates the magnitude of the work effort required to implement a DAL.

Database Table Count	CRUD SP's	Indexes (RUD) SPs 3 or 7 per table	Data Object Functions (Code Functions - C#, VB)	Web Endpoints (Code Functions - C#, VB)
Small 5/20	20/80	15/60 to 35/140	35/140 to 55/220	35/140 to 55/220
Medium 20/40	80/160	60/120 to 140/280	140/280 to 220/440	140/280 to 220/440
Large 40/200	160/800	120/600 to 120/1400	280/1400 to 280/2200	280/1400 to 280/2200

Table 1 - DAL Sizing

As this Table 1 depicts, the work effort required to design, develop, implement, and manage a DAL for medium and large organizations is nothing less than daunting. Depending on the complexity and requirements for the stored procedures, they can range from 30 lines of SQL statements to hundreds based on the implementation of transaction control, error handling, parameter definitions (data in/out) and documentation. The coding effort for each stored procedure to be exposed via the DAL Object layer is considerable as each function (or subroutine if nothing is being returned) requires defining every input and output parameter, they require connections to the database, error handling and lastly, documentation. The final layer, Web Services can also be as cumbersome as every Data Object has a corresponding endpoint Service and interface handler to expose the Data Objects via Web Services.

See http://www.konigi.us/Documents/Tiered%20Architectures%20and%20Data%20Access%20Layers.pdf for additional information on the background and the magnitude of a Data Access Layer within a tiered architecture.

To further complicate this scenario, for every change in the data model or the mapping of columns to support the BLL, each point of access of the 3 DAL sub-layers may require modification and regression tested prior to providing the DAL to the developers. If the DAL Web Endpoint service consists of both REST and SOAP, then potentially double the number of services and endpoints required; the DAL Data Objects are utilized by both SOAP and REST if the architecture is properly designed and implemented. Depending on your final coding design/architecture, your DAL for a medium to large database could encapsulate 200k-1M lines of code and scripts. From the table above, the number of high-level language (C#, VB) functions between the DAL Objects and the Web Services consist of at least 560 to 4400 functions (DAL Objects + Web Endpoints).

This is only part of the DAL story. Since CRUD and Indexes impact a single table, developers most often require data representing multiple tables which is accomplished using query joins and union statements. This requires customer developed stored procedures and can be complex depending of the developer's requirements. The "customized" stored procedures impact the necessity to also develop views and functions to support queries and many of the views may also need to be exposed via the DAL. These elements make up the inherent complexity of a DAL and is typically driven by a combination of the developer's needs and the DBAs scripting efficiency. A single query returning columns from multiple data tables is common for populating reports, lists, data grids and other UI controls. The same requirements are also prevalent with regards to updating, inserting and deleting data. Some DBAs and developers utilize stored procedures to perform a myriad of tasks including selecting data across multiple tables using views, functions and other stored procedures. They utilize stored procedures to perform multiple updates, inserts and deletes to the point that stored procedures

become so complex that they represent functionality more akin to a Business Logic Layer than a data repository. The problem with this approach is that Business Logic rules tend to reside in multiple realms, within the BLL, the DAL and potentially in the UI. In such cases as these, a DAL can be more of an impediment than a benefit. Personally, I highly recommend keeping a database agnostic to BLL rules and keeping the DAL as atomic as possible (1 query, 1 table for CRUD/RUD). Updating multiple tables with a single stored procedure is not a good practice as is returning multiple result sets with a stored procedure. This strategy of stored procedure development requires complex DAL Objects to segment and manage the responses of stored procedures. This turns the database into another development environment, contaminated with BLL rules. Once again, Developers are not DBAs and DBAs are not Developers; let us keep the capabilities of resources specific to their roles!

The outcome of this architecture is that a DBA must work to resolve business logic related issues or open the database up to developers to "do as they please" which negates a DAL altogether. Simplicity and normalization are key concepts to designing and implementing a DAL. If stored procedure return multiple result sets or if they update, insert and/or delete multiple records with a single execution or even worse, if they accept dynamic queries and can dynamically change different tables depending on the parameters, then utilizing these types of stored procedures within a DAL should be strongly avoided. All too often, DBAs and developers transform a database into a high-level development environment. One individual I collaborated with actually built a fully operational solution using stored procedures, views and functions and the output was HTML which was sent to a browser via execute statements. No compiler, no high-level coding environment, just a SQL database.

A well-designed and architected DAL exposes Table CRUD & RUD, Custom Stored Procedures, and Views. Each Custom stored procedure, function, or view requires a corresponding DAL Data object and a matching Web Service endpoint. Let the developers consume/publish data via the DAL Web Service endpoints of inter-mixing technologies, roles and logic across your solution.

THE SOLUTION

How do we implement a DAL without creating another IT department (which some organization do with Data Architects)? The solution to implementing a comprehensive Data Access Layer is to either develop all of the layers (Stored Procedures, Class Objects, Web Services) using a combination of manually written and/or automated methods or acquire technologies that can help to generate some or all of your Data Access Layers.

MANUALLY DEVELOPING A DATA ACCESS LAYER

The most efficient method to develop a DAL is to template and develop each layer's code/scripts for each of the CRUD elements as a baseline. Include relevant error-handling, responses, object definitions starting with the Stored Procedures as these will bubble-up to the subsequent layers. Include handling for changed rows, transaction control, row count limits and any logging required for the database data access layer. Oracle and SQL Server have relatively identical footprints and functionality with regards to parameters and definitions yet have different handling for errors, transaction control and responses syntaxes. I opt to include all field names for

queries which can be time-consuming with tables consisting of a large number of columns (typically denormalized tables). This will improve readability and documenting of the scripts. There are numerous examples of templating stored procedures especially in the Microsoft realm including some interesting examples that utilize T4, a textual and coding template toolkit that can generate code or scripts based on patterns. I personally have used T4 for some lightweight routines but found it easier to develop code generation functions in native languages such as C# and VB. Using a scripting language to develop both compiled code and SQL scripts proved frustrating and time-consuming as the entire, functional set of controls and objects inherent in Visual Studio is not available via T4 which is critical to support user-entry, option selecting and visual feedback of routine activity. Yes, you can make calls to DLL's with high-level, compiled code yet why implement multiple technologies when a single language will suffice. Do not overly complicate a DAL with multiple scripting languages (T-4, Powershell) and a high-level language (C#, VB). Debugging and testing will be extremely complex and slow.

Coding of the upper two layers, DAL Object and Web Services becomes more complicated if you are supporting more than one database as syntax, objects and formatting of each environment needs to be observed and implemented properly. Datatype matching is another issue as Oracle and SQL have variations in their datatype definitions. Again, prototype each example for the class objects and Web Services for each CRUD element exposed by the stored procedures. The result will be a useful and stable set of templates to support your CRUD access. Next, expand into the scripting and coding to handle Index manipulation which will complete the first steps for architecting a versatile and flexible DAL. To support our initiatives, we developed numerous routines with a myriad of formats including implementing functions that returned XML, JSON, Datatables, Datasets, defined and undefined objects, classes and every available datatype, by database (Oracle and SQL Server) including implementing List(of) methods. We included parameters being passed to the functions as well as returned to the Web Service/Interface routines. Remember this is a foundation design so include as much functionality and consider the worse-case scenarios for your data requirements, within reason. This last point is key to efficiency and optimization of your DAL. The more cumbersome and convoluted, the more difficult it will be to diagnose and enhance your DAL if future modifications are required.

One framework we evaluated was to have all datatype and formatting of database objects be handled with DbProviderFactories which is part of Visual Studio Active Data Objects. These classes for this framework level sets the implementation of data objects. This proved ineffective as the native datatype for Oracle and SQL Server are different than those exposed by DbProviderFactories and to ensure concise and functional implementation of fields and parameters from the stored procedures through the Data Access Web layer, we identified the necessity to maintain a single point of reference to element mapping and datatype naming. DbProviderFactories masked the Oracle elements to .Net Data elements which are not useful when developing PL/SQL scripts for the stored procedure creation. Considering maintaining isolated datatype mapping and controls respective to each database's particular element definitions when building your DAL.

Other things to consider are deploying these base templates to support solutions. The point of this is to expose database data to the BLL or other consuming stakeholders. These templates need to be as atomic as possible and readily duplicatable. For our models, we

minimized the first layer defining stored procedures to optionally include transaction control, row count restrictions, error-handling/logging, and returned rows changed. These 4 options provided the most useful variations for stored procedures and have been the focal starting point for our DAL implementations.

Once the Database Data Access layer is completed, fully test the routines extensively. This can be accomplished with either code templates or via a number of off-the-shelf tools that can call stored procedures. Build out worse case scenarios for your data including handling of nulls, attempting to duplicate primary keys, deleting constraining records, using incorrect data types (sending strings to numbers, decimals to binary, etc.). flush out all the issues and enhance your Database Access Layer stored procedures to accommodate any issues. This is a useful point to use templated code in your Data Object layer to validate, log and perform any transformations necessary. A useful tip is to include in your Data Objects layer any functions necessary to transform data as this will become necessary once your team needs to change from XML to JSON or strings or arrays. Depending on how you normalize your initial stored procedures calls, you will need to reformat the response objects to other object-types to accommodate variations of output to support your Web Service layer. I found that emitting either XML or Datatables as the base for all stored procedures responses was useful as both are serialized and can be easily transformed into other objects with simple functions. As noted, we are templating at this point and developing a versatile architecture will result in a framework that can accommodate almost any data access requirement.

The Data Objects region is also the most important layer to interact with the OS so include any event log and error handling here. Also consider any SMTP notifications or ITIL submissions as you can template them via XML support files and change them if required without change the underlying code. Another consideration would be dashboard integration, data governance elements, subscription validations, token control or name mapping such as noted previously with the “Data Model Layer”. This functionality can be readily implemented in the Data Objects layer so evaluate and consider implementing them at this point.

Lastly, Web Services, the final layer and the most visible to the development teams and/or external stakeholders. Collaboration with the development team and/or external consumer/publishers is necessary at this point to ensure interoperability and transparency with exposing the DAL Objects via Web Services. Elements and technologies such as IAM, logging, encryption, transport methods, encapsulation and coupling should be considered, prototyped and evaluated. If you are sharing with external consumers, then IAM become important as does encryption if your data is sensitive. If you are sending via the Internet, consider all possible precautions to publish and consume data. For most implementations where data is consumed by internal applications, IAM should be implemented at the Web Service layer and this can include LDAP, federated models (SAML) and others. The Data Objects layer is already coupled to the Database Data Access layer so authentication here is limited to the Execute Only account provided by the DBAs. If you are designing and deploying an integrated solution (include Web, and DAL Objects) then there is no need to have an IAM or additional authentication framework between the Web and DAL Objects. If you are designing decoupled Web and DAL Objects, then consider another level of IAM between these two. I found coupled Web and Data Objects less cumbersome, easier to debug and deploy and since they are fully integrated, with IAM implement at the service connection. One other topic that is worth mentioning; many architects and developers believe that authentication needs to transit the entire DAL and be validate by the database. I disagree with this principle as I have

consistently maintained and recommend a decoupled framework and I leave the DAL Objects to Database Access Layer security to the "Execute Only" account. Including credential validation via the database for the consumer (application side) transforms the database into an authentication point; not a very useful capability considering other IAM control points such as the OS or a Federated Security application may already be supporting that functionality. Let the endpoint handle interrogating IAM if necessary and once validated, the data can pass seamlessly. If your architecture include ESB, a security appliance or other types if API management, they can also accommodate IAM or transport security controls. Keep the DAL performing its intended function, the movement of data seamlessly and utilize your other security environments in the final layer of your DAL.

The Web Service layer can provide numerous opportunities for almost limitless usages. I have collaborated with clients where DAL Web Services enhanced Dashboards, communicated with State and Federal agencies, transformed peer and partner systems, integrated acquired technologies, supported business objects (reporting, analytics and big-data), supported mobile apps, coalesced systems and too many other solutions. This level of exposure for your DAL provides virtually limitless possibilities all of which apply best practices for your data access and control. Your Cybers-Security groups can monitor and react to payload elements, BPM and MDM activities can be seamlessly driven by submissions, infrastructure teams can efficiently monitor and control resources to support the DAL, and lastly the Database team can manage, scale and monitor the underlying repository confident that restrictions and implemented architectures will ensure data consistency and that security is maximized while dissemination prospects are minimized.

ACQUISITION OF A DATA ACCESS LAYER TECHNOLOGY

The other option is to acquire a DAL solution. There are variations and partial solutions yet none of these solutions apply database best practices for querying/accessing data or expose data via a three-layer architecture. The critical first layer, the Database Data Access should fully implement only Stored Procedures and this perception can be readily verified by searching the internet for Database Data Access Layer Best Practices. Oracle and Microsoft have the two most implemented enterprise databases and each organization has respected professionals who have cited the importance of utilizing Stored Procedures to minimize SQL Injection, maintain query best practices, optimize the database engine, and to limit malicious data access attempts. Additionally, there are hundreds of white-papers, architecture diagrams, books and publications that focus on implementing Data Access Layers and the underlying fundamental principal noted in all of them are the necessity to implement Stored Procedures.

At Konigi, we have developed technology that fully generates a Data Access Layer across all 3 layers, the Database Data Access, DAL Objects, and Web Services. The solution starts by creating and implementing the CRUD/RUD Store Procedures, one for each table and one to three stored procedures for each index (if you choose Update and Delete by Index). The solution generates the DAL Objects (.Net Classes) which executes the Stored Procedures. Finally, the solution generates the Web Service endpoints (Service and Interfaces) to expose the DAL Objects. This capability includes exposing customer/custom developed Stored Procedures, and Views. The solution is ***DAL_GEN***, Data Access Layer Generator, designed to integrate with Oracle's Database and Microsoft SQL Server Environments.

DAL_GEN is a comprehensive tool for generating, building, and deploying the layers necessary to expose Oracle or SQL Server databases across all three Data Access Layers: Database Data Access, DAL Objects, and Web Services.

DAL_GEN generates and implements Stored Procedures scripts with a set of optional settings to support error-handling, transaction control, MAX RowCounts and a number of other configurable options. Prefix and Suffix labeling of generated stored procedures along with filtering of tables is also provided. T-SQL and PL/SPL scripts are generated and executed automatically. They are consistent, utilize all table Column names in addition to table Index and Key names for parameters and all stored procedures are documented. Re-generation of a DAL_GEN project removes old/orphaned Stored Procedures automatically, keeping the Database Data Access portion along with the generated code of the DAL consistent and manageable. Stored Procedure Suffix and Prefix allows for segmenting large databases with multiple DAL_GEN projects.

DAL_GEN generates the DAL Data Object code using Microsoft's a Visual Studio solution template. This layer executes the Database Data Access Stored Procedures. The generated code creates functions with parameters mapped to stored procedure parameters along with their respective datatypes. This eliminates the requirement for developer coding to access the generated DAL stored procedures. Each DAL Data Object function is consistently templated and created as a pattern. The DAL Data Objects implement ADO calls to the DB using pre-defined Username/Password authentication. The selection of function output is selectable as stream (JSON), string (XML), Database, or Datasets. The most important aspect of DAL_GEN is the coding of the DAL Data Objects is eliminated as a requirement of the Development teams or the DBA's. The generated DAL Stored Procedures, Data Objects and subsequent Web Services are fully functional and compiled. No developer or DBA interaction is required once the solution has been templated and configured within DAL_GEN.

Lastly, DAL_GEN generates Web Services via Microsoft Windows Communication Foundation (WCF) endpoints. User-selectable endpoints for REST (XML, or JSON) and/or SOAP (XML, JSON, Datatables, Datasets) allow for virtually any configuration of data consumption. A single project can expose SOAP, REST or both SOAP and REST, if required. All changes to the Visual Studio project, the Web. Config file and the solutions related publishing files are fully controlled by DAL_GEN including compiling, and publishing to the target IIS web server.

DAL_GEN is the only solution on the market that completely generates every CRUD and RUD SQL Stored Procedure, Class Objects to access the Stored Procedures and Web Services to provide endpoints to the developers to consume "Data as a Service".

DAL_GEN provides capability to for compiling and publishing solutions to an IIS web server for the immediate implementation by developers or consumer. DAL_GEN supports both Microsoft's Azure and Amazon Web Service VMs and SQL Server/Oracle databases, either cloud or local. Solutions can be hybrid hosted/cloud or locally provisioned. The best part: the scripts and code generated by DAL_GEN are fully usable scripts and source code; no DLL's are required to utilize them. Everything element created by DAL_GEN is 100% transportable, readable and ready to implement. DAL_GEN generates the Database Data Access stored procedure scripts,

generates the Visual Studio Source code, updates the VS project, Web. Config and related files and implements user-selected publishing methods to promote the compiled code to TEST or UAT servers.

Database model changes can be facilitated in “seconds” not days, weeks or months. Changes to database Tables, Custom Stored Procedures, or Views can be ready via regeneration and deployed rapidly by DAL_GEN publishing to your IIS site, efficiently and consistently. The Web Service endpoints are visible once publishing is complete. DAL_GEN also provides YAML for REST endpoints and WSDL’s for SOAP documentation. At this point, expose the Web Service endpoints to development team, your ESB, API management environment and then you are able to control, monitor and govern your data “as a service”. If you want to change from XML to JSON, change a checkbox, regen the project and in seconds, JSON output. Datatables, Dataset, again, set a checkbox and regen. Developers simply update their web service reference and all endpoints are refreshed. DAL_GEN can even provide strongly typed dataset via XSD via SOAP services. Now you can implement strongly typed data opposed to having to hard-code each data element.

Remember, an important element to consider for manually developed DAL’s is the magnitude of code and scripts required to design, develop, and implement a fully architected DAL environment. As mentioned in “WHY ARE DATA ACCESS LAYERS NOT COMMONLY IMPLEMENTED? WHAT ARE SOME OF THE OBSTACLES?”; the number of store procedures, functions, and web services that constitute a multi-layer DAL are extensive and require constant enhancements for each data model change. This equates to financial costs, resource allocations, delays, and numerous points of failure inherent with developer coding practices and styles.

If your organization implements Oracle or SQL Server and enhancing data security, reducing development costs, provide consistency for coding of data access objects, and managing, controlling and monitoring your data are strategic initiatives, then contact us at Konigi.us for a unique solution.

For more information or a demonstration, contact or visit us at www.Konigi.us