

The SOA Magazine

Feature Article



Web Service-Enabling Relational Databases for SOA

by Robert Schneider

Published: October 23, 2006 (SOA Magazine Issue II: November/December 2006, Copyright © 2006)

[Download this article as a PDF document.](#)

Abstract: For architects and developers in search of quick wins to demonstrate the value of a service-oriented architecture, databases present a largely-unexplored landscape of vital information. Although previously time-consuming to integrate, new tools and technologies make it easy to service-enable these information silos. The result is an elegant, cost-effective infrastructure that preserves IT investments while facilitating modern application development in support of service-orientation.

Introduction

Despite the continuing rise in service-oriented architecture acceptance and adoption, many business users, developers, and systems administrators are still seeking real-world applications to highlight its power and flexibility. Luckily, just about every enterprise has a valuable collection of information that's just waiting to be brought into the modern, service-oriented era. And to make things even easier, most of these assets are neatly organized within databases.

In this article, I describe how you can leverage legacy corporate intelligence to provide immediate value to your user community by incorporating and exposing it as part of an SOA.

Databases in the Service-Oriented World

Way back at the dawn of the relational database era, applications were often tightly coupled to their underlying information repositories. There were many reasons for this, including the often proprietary APIs sported by each database platform, or, believe-it-or-not, the myriad of homegrown database engines built and maintained by IT shops. This caused no end of trouble for application developers and software vendors, who had to maintain different versions of their solutions for each major database platform.

Things began to improve in the 1990s with the rise of the Open Database Connectivity (ODBC) standard. Since 1995, this technology has been included with the SQL/CLI (Call Level Interface, based on Microsoft's ODBC 3.0 and CLI) standard maintained by X/Open and ISO. This helped developers write one database-independent application, decoupling their business logic from their data storage logic. Modern SOA implementations take this approach to the next level.

When examined from a database-centric viewpoint, a service-oriented architecture seems like another logical step towards the ultimate goal of fully decoupled, service-based computing. Of course, SOA offers a much broader set of benefits for information sequestered in databases.

Before beginning the discussion of incorporating databases into this new architecture, it's worth noting that along with contemporary SOA platforms comes a renewed emphasis on representing data with standardized XML. However, even though it's tempting, you should resist carrying this approach all the way down to the physical data level (i.e. the database). Otherwise, you might find that your physical data model coupled too tightly with the service contract,

thereby negating many of the architectural benefits of SOA. We'll review how to apply good design patterns for this type of task a little later in the article.

As the primary data storage mechanism for both homegrown and packaged applications, relational databases represent a tremendous investment. Anything that we can do to lengthen the life and utility of these systems and the data they house, helps preserve this investment.

Incorporating your database into your overall SOA is an effective means of exposing these assets to a much wider audience. In general, making an existing database SOA-ready is non-invasive: You shouldn't need to alter any existing tables, views, or stored procedures. You may, however, elect to create some new objects such as views or stored procedures to help support your initiative and reduce (perhaps even nullify) any hand-coding or other time-consuming labor.

Because you're not changing the structure or behavior of any existing objects, there's a very good chance you won't break anything that already works. Alas, since Murphy's Law was likely written with software developers in mind, you'd be wise to back up all aspects of your production platforms, as well as conduct your development efforts in a separate environment if at all possible.

Rather than making an all-or-nothing effort, you can bring your databases into the SOA world in a manageable, phased approach. This evolutionary, iterative strategy lets you make adjustments on the fly, yielding better results, while helping you keep your sanity intact.

Before we move on to exploring how to actually guide your database through this transition, just a few words of caution:

- *Security:* Just because you elect to create a service façade to encapsulate your database doesn't mean that you have to throw security to the wind. Exposing data that was previously accessed by a limited and controlled user base can introduce a variety of security concerns that need to be addressed and accommodated. Fortunately, there are many security mechanisms at your disposal, several of which can be combined to facilitate access of the same information set by numerous service consumers.
- *Performance:* You should be mindful of the potential negative impacts on performance that "opening up" your database can entail. For example, your production database hardware and software platforms may have been configured with a relatively limited number of users in mind. Suddenly expanding this audience can drag down system response while concurrently filling up your email and voicemail inboxes with unprintable greetings from aggravated users. This is yet another reason for choosing a conservative, evolutionary path towards your goal.

To witness to the marriage of SOA with your relational database in the real world, the following section looks at a simple scenario. (Note that I won't spend any time in code; as you'll soon see, there are now technologies out there that obviate the need for coding. Instead, I'll stay at the conceptual level and keep things simple.)

Making it Happen

Congratulations! You've just accepted a job with a small, yet progressive-minded IT shop. Your new employer is "Pay-N-Pray Motors", a deep discount rental car operator. To keep costs down and profits up, they offer their customers a selection of "legacy" automobiles, many of which are older than the people who rent them.

To keep on top of the ever-changing rental car market, Pay-N-Pray will soon be offering a self-service Web site that will likely be utilized by their most frequent customers. Much of the functionality that's needed is already present in a client-server application that was built some time ago.

To begin, the Web site will need to offer a relatively small set of capabilities. Customers must be able to view information about their accounts, as well as update certain key details such as their address, phone number, and so on.

As a way to reward the most frequent customers, management also wants you to implement a page where these valued clients can register interest in an exciting (albeit low-cost) collection of Pay-N-Pray branded gifts such as golf tees, ballpoint pens, notepads, and so on. Since these gifts are reserved for only the most valued customers, there's great concern that lower-end clients might try to register for these fabulous benefits. You'll need to make sure that this doesn't happen by either writing new business logic or leveraging something that already exists. In terms of budget and schedule, naturally things are very tight. *You won't have much time or money to get the job done.*

As you begin your investigation of the existing application, you learn a variety of facts. Some of this information is encouraging, while the rest is rather depressing.

First, the client-server application sports a gnarled mess of spaghetti code. From what you can tell, there is minimal separation among the user interface, business rules, and database access logic. You would ask the original developer for some guidance, but he was fired 6 months ago after asking for a raise.

Naturally, there's no API to leverage. However, given the tangled muddle of the application code, it would have been hard to create a clean programming interface anyway. However, after all of the above bad news, you're gratified to learn that the application uses a real relational database. On top of that good news, it appears that the system developers consulted one of the many fine database design and tuning books on the market, yielding a well-thought-out logical and physical data structure. Happily, the database offers both stored procedures as well as robust ODBC drivers.

Finally, in talking to your manager, you learn that she's open to both .NET as well as J2EE technologies. This takes the religious war off the table: You're free to select whatever approach makes the most sense for your environment.

Comparing the Options

The first option would be to use one of the database-driven Web service generation tools on the market to quickly crank out a full set of CRUD (create, read, update, and delete) Web services for each table. You would then use your application development platform of choice to interact with these new services.

At first glance, this seems like an ideal approach: It's fast, easy, and inexpensive. Be aware, however, that there's danger in this strategy: These Web services will not have standardized contracts and will bypass much of the business logic that your existing application might have in place to protect data integrity, enforce business rules, and so on. Objects such as triggers will still work; the Web services won't be able to bypass them. However, in this example, all of the existing business logic is intertwined with the user interface and database access code.

If the first approach is the lazy way to SOA, then the second tactic is bound to get you off the couch. To give yourself maximum power and control over your Web services, you could elect to hand-code them against the ODBC API and then work with these services with your application development tool. There are many examples of how to write this code available on the Internet.

The beauty of this method is that your new Web services could include whatever business logic you want to incorporate, and would be custom-tailored for your environment. With each service you deliver, you have the opportunity to apply standards and keep it in alignment with other services, a fundamental success factor to achieving some of the bigger benefits associated with SOA. Now for the bad news: hand-coding always takes longer and can introduce all sorts of debugging and maintenance challenges.

There's always the compromise of allowing some Web services to be auto-generated while ensuring that others (especially those that encapsulate stored procedures) are best hand-coded to preserve and express important business logic in a consistent manner. Following this road would require you to analyze the exact requirements, and select the correct technology on a case-by-case basis.

Service-Oriented Principles and Databases

While you could just slap some Web services together and call it a day, you would be wise to attempt to follow some of the common principles and best practices for service-orientation. Otherwise, all you're doing is simply exchanging one messy environment for another. When evaluating an SOA interface to your database, here are the key principles [REF-1] to keep in mind:

- *Service reusability*: Management and technical talent rarely agree on anything. However, the reusability aspects of a service-oriented architecture are one area in which both sides generally concur. To derive maximum reuse from your database-driven services, it's crucial that you keep their operations as general-purpose as possible. For example, you might elect to create a generic query operation that expects a collection of input parameters. A generic operation would be able to run even if only one parameter were supplied. Multiple parameters would simply reduce the result set.

- *Service contract:* Given the ease with which you can their structure, databases are particularly vulnerable to contract obsolescence. This can cause havoc when service consumers have been coded with a fixed contract in mind. You can mitigate this risk by tightly restricting database changes as well as building more fault-tolerant clients. A useful strategy to employ when database changes are unavoidable is to use objects such as views and stored procedures to shield the consumer from changes. Finally, constructing fault-tolerant clients can serve as a robust last line of defense.
- *Service loose coupling:* A well-thought-out service contract will contain all information necessary to instruct the service how to behave. This is very common for auto-generated, database-driven Web services. For example, an INSERT operation's contract will typically specify all parameters necessary to create a record. Any service consumers can review the contract and understand what information must be passed to the service.
- *Service abstraction:* This concept is best illustrated with a data validation example. Suppose that you have a service that accepts a set of input parameters and then combines some business logic and table lookups to check that a condition has been met. By segregating the business logic into the Web service (rather than requiring the consumer to apply its own logic), you're free to make future changes to the service-based business logic with confidence: These alterations won't break anything in the consumer.
- *Service composability:* For databases, this trait, which refers to the ability to combine multiple services into a higher-level service, is often addressed by grouping several stored procedures into a higher-level procedure. This higher-level procedure may itself be converted into a Web service. When evaluating the decision on where to create your database-driven Web service and how to orchestrate its behavior, you should factor in concepts such as reusability and autonomy to help you decide the optimal mix between Web services and stored procedures.
- *Service autonomy:* For this concept, your goal is to ensure that each service has minimal dependencies or other entangling requirements from other services or resources. From a database viewpoint, you must also expect that your services will share the database's resources. This type of autonomy is known as service-level autonomy.
- *Service statelessness:* In most cases, your goal should be to avoid maintaining state within a given service. The good news when designing a database-driven service is that you have a wonderful persistence platform at your disposal: the database itself. Aside from persisting information in the database, you should strive to keep your services as stateless as possible.
- *Service discoverability:* It's a good idea to employ a repository strategy to help register these services along with their purposes, especially if your enterprise has many developers. Data services (or services that encapsulate data access capabilities in a generic manner) are prime candidates for high reuse. They address a fundamental cross-cutting concern and should therefore be easily discoverable.

The Solution

Armed with a better understanding of how service-orientation relates to data services, as well as the benefits and tradeoffs of each of the previously describe approaches it's time to look at a solution for our scenario.

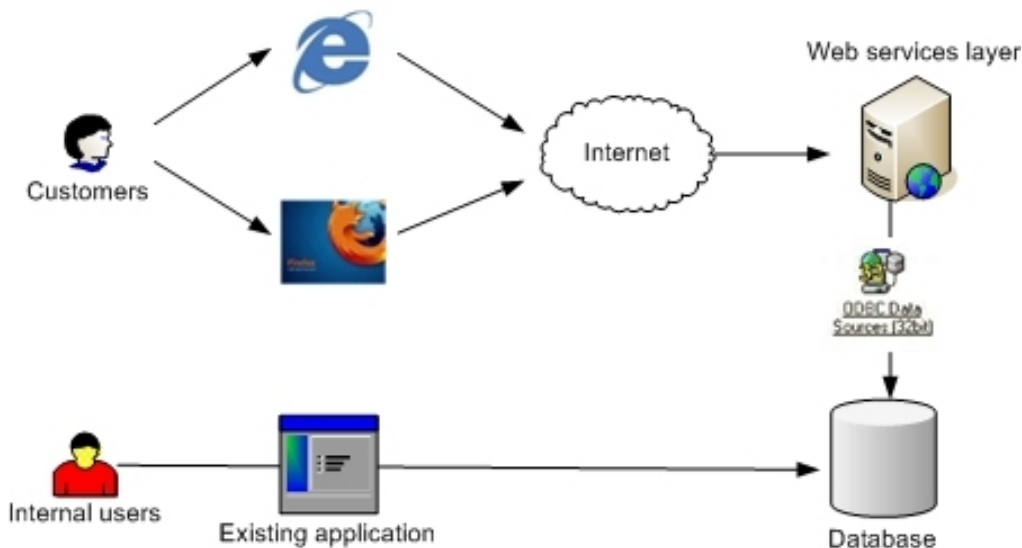


Figure 1: Architectural components associated with the Pay-N-Pray Web site and its underlying services.

Factoring in the time and especially the cost restrictions imposed on us by Pay-N-Pray management, the choice becomes clear: the hybrid approach (a mixture of automatically generated and manually created services) will deliver the fastest architecture while still establishing standardization for some of the more important services.

Figure 1 displays all of the moving parts, while Figure 2 (below) shows how the new services abstract underlying data structures. In Figure 1, observe that external users (i.e. customers) access a Web services layer that interacts, via ODBC, with the database itself. Internal users (i.e. employees) continue to use the existing application to work with data. In Figure 2, note that for this example I've kept a one-to-one ratio between services and operations.

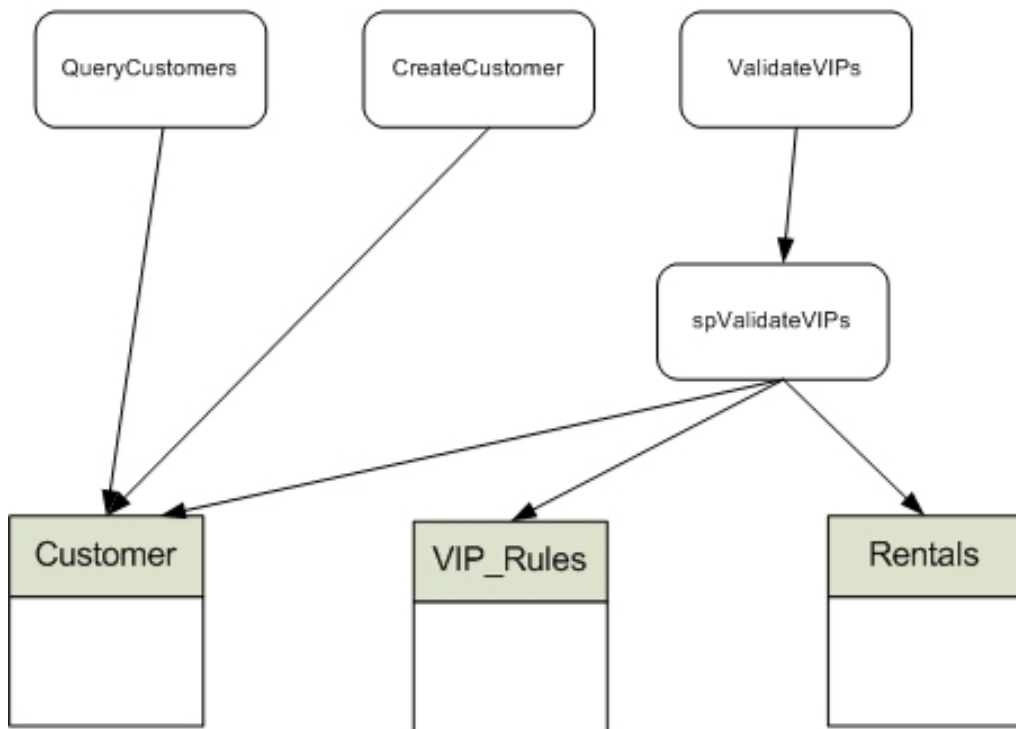


Figure 2: The relational data structures abstracted by Web services.

To begin, you elect to employ a third-party product to generate read-only Web services on information such as address, account history, and so on. Some of these products generate application code that you must then compile and publish; others go the extra mile and fully generate and then publish the Web services onto a server. The server then speaks ODBC to the database engine.

The *QueryCustomers* operation is an example of an automatically-generated service. If you find that you need to perform some data aggregation or joins, you could create views on the database server that span multiple tables. Many of the Web service-generation products will also create services from views.

For certain types of operations (such as determining whether a customer meets the rigorous standards for the free prizes), you will need to create Web service-ready business logic. In this example, other than using it as a reference, there's no point in trying to untangle the business logic from the current application. However, one way to provide higher levels of abstraction, loose coupling, and reusability would be to take advantage of the stored procedure language offered by the database. Since logic written in these stored procedure languages is often more streamlined than traditional application programming languages, you are also likely to significantly reduce the amount of new application code necessary.

With your stored procedures in place, you can then use the same generation technology to turn these new procedures into standards-based Web services. In Figure 2, I've created a stored procedure called *spValidateVIPs*, and then Web service-enabled this procedure. The procedure contains all necessary business logic to ascertain if a particular customer is eligible for the fabulous premiums described earlier.

Finally, operations that create new records or update existing ones may either be hand-coded or automatically generated. The proper approach is highly dependent on your data and business rules. In many cases, creating a new record entails all sorts of validation and integrity checks, which would argue for more control over the Web service. On the other hand, updating non-key fields can often be done by an automatically generated Web service. In Figure 2, the *CreateCustomer* service is hand-coded because there are many business rules that need to be enforced before allowing a new record to be created. This necessitates either a stored procedure or a new Web service.

Conclusion

Armed with your collection of newly-minted database-driven Web services, you can quickly construct and deploy the browser-based application. You've also created a solid foundation for leveraging existing information and investments. Even better, you've done it with balanced effort while complying with solid standards and best practices.

References

[REF-1] <http://www.serviceorientation.org>

THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL



[home page](#) [past issues](#) [contributors](#) [official book site](#) [legal disclaimer](#) [rss](#)

Copyright © 2006 SOA Systems Inc.

All Rights Reserved