

RPG and the AJAX & SOA Technology Trends, Part 2

By Ted Trichew

Software Architect with Databorough

Ted has over 20 years of software development experience, including 16 years at IBM Canada Laboratory in Toronto. At IBM Ted worked on mid-range software, including architecture and development of original releases of C/400 and ILE/RPG. Ted also has 5 years experience as product manager of the Ironside Business-to-business solution suite which provided one of the first rich Internet user interfaces available on the market, and also provided integration in iSeries ERP systems including JDE World, BPCS, PRISM, MAPICS and JBA System 21. His most recent experience was at SSA Global as an architect working on the SSA Technology Architecture platform which provides a modern rich user interface, as well as service oriented architecture for SSA's solutions. Ted has a thorough understanding of the issues facing businesses that want to modernize their legacy applications.

Abstract

Two technology trends that have caught the attention of enterprises, development tool vendors, and industry analysts because of their potential business value are AJAX and SOA (Service Oriented Architecture). In part 1 of this article, we discussed AJAX as a technology that brings the promise of client/server type rich user interfaces to the web. We concluded that the key to unlocking the promise of AJAX lies in leveraging of legacy business logic and data. In Part 2 of this article we will now examine the service Oriented Architecture (SOA) technology trend, and how it can go hand-in-hand in the creation of a modern application architecture that provides common access to your legacy application business logic and data.

Legacy Integration & SOA

One technology trend that has been hyped as having the potential to solve legacy application integration is Service Oriented Architecture. As the name implies, this is not really a product, or framework that you can purchase, but rather an architectural pattern for abstracting the business processes in an organization into business-oriented services.

What makes up a SOA?

There are four main components in service oriented architecture:

1. *Transaction Services*: These are services for automating the business process. For example, "create quote" would be a common order management process service. Transactions services are what most people think of when they talk about Web Services and this is why many people equate SOA with Web Services.
2. *Data Services*: These are services that allow you to read and display business data. An example would be querying the status of a quote.

3. *Services Registry*: The registry provides a way to publish and discover the re-useable Transaction and Data Services. All important information about the service, including its name, input/output parameters, and where it can be found are stored in the registry.
4. *BPM/Orchestration*: Provides a mechanism to tie together the services and coordinate interactions among services. This is the tool that the developers will use to create new business solutions by tying together existing services.

Why SOA?

The main point of SOA is that it provides a way of abstracting the business logic and processes in a business in such a way that you can deploy a set of shared re-useable services. Typically this “business service layer” will contain coarse grained business service interfaces while the legacy application logic and data sit in an “infrastructure layer” that provides much finer grained interfaces that are leveraged by the business services layer. The business service layer would contain “generic” transaction and data services such as “create order”, “create quote”, “get open orders”, “get product information” etc. These services are implemented by integrating with existing business logic and data in the infrastructure layer. In general you are accessing legacy applications that can be on one or more back office systems. For example, you may be retrieving financial information from a Financial Management system interface, and you may be submitting an order through an ERP system interface.

SOA helps with AJAX

If you are considering implementing a new rich client application which uses AJAX technology, as discussed in part 1, then having an SOA will greatly reduce the effort. The reason for this is that rich client applications typically have multiple layers. You would have an “application UI layer” which contains the rich UI and any UI event handlers that handle interactions with the UI. You would also have an “application logic layer” that provides a set of application specific UI “services” or api’s. For example, if you are creating a new AJAX based self-service ordering system; you would have a rich UI that contains a “Self-service Ordering Screen” and associated event handlers for handling users key strokes as they interact with the rich UI. You would also have an application layer which contains application services such as “get self-service product information” that is triggered when the user enters a product number, and then presses the tab key; or “create self-service order” that is triggered when the user presses the “complete order” button on the rich UI.

We need to create an SOA Layer

In order to implement the application services, you either would integrate directly with the Legacy ERP system, or you would go through an SOA business services layer which provides generic business services such as “get product information”, and “create order”. If you go through an SOA business service layer, the application layer is more of a mapping layer which maps between the generic business services and the actual UI application services. For example, you may have a “getShipTo” transaction service that returns the user’s ship to information as one big string of information. However, the

applications user interface requires that the ship to address is split into “Ship to Number”, “Street Address”, “City”, “State”, “Country”, and “Zip”. An application service for getting the shipTo would have an interface that provides all the individual parts of the ship to address, and would be implemented by parsing the results of the call to the “getShipTo” transaction service in our SOA layer.

The business services layer in our SOA provides us with the most flexible solution, from the perspective of being able to handle future changes to business processes. So essentially we provide a generic set of business services such as “create order”, “create quote”, etc. that many different applications an access to provide their own application specific interfaces or services such as “create crm order”, or “create ecommerce order”. The generic services would be implemented by accessing legacy application logic in the “infrastructure layer”, like “create JDE order”, or “create MAPICS order”. This means that all the interactions with the legacy system are isolated in a single place, rather than being scattered around many applications. Since the generic layer reflects the company’s business processes, rather than the legacy system implementation, the user interface of new applications can be more reflective of the actual business process of the company, rather than the legacy systems implementation.

Creating Composite Applications

One other great advantage to providing a business service layer and one of the promises of SOA is that you can create new “composite” applications that are made up of multiple disparate transaction and data services. For example, I may have an application that allows for self service ordering by taking advantage of ERP order management, but also allows the user to view their accounts payable available through a Financial Management system. As the application developer, I don’t have to know what systems I am talking to, I just need to access the order management related services, and the financial management related services that are published in the “services registry” in the SOA. I can weave these services together into a new composite service the Business Process Management tools provided with my SOA deployment. For example: IBM’s *WebSphere Integration Developer*.

Re-using existing application logic is the reason

It all sounds great, but there must be a catch. In order to truly unlock the value of AJAX and SOA and make an organization more flexible and agile, we must be able to reuse existing “legacy” business logic. This is not always as easy as it sounds, as many organizations have found when they try to integrate with legacy iSeries applications.

Most legacy RPG applications were written in an era where developers did not layer their applications in order to separate their user interface from their business logic and data. More “modern” applications adhere to the architectural principal of “separation of concerns”. That is, they separate the user interface, the business logic and the business data in order to make applications more modular, and maintainable. If we use a food analogy, one can think of the architecture of a legacy RPG application as a tangled pot of spaghetti, and what we really need is a more layered approach like a plate of lasagna.

Since existing RPG programs are typically monolithic with repetitive code across programs, it makes it very difficult to easily access the business logic that we need in order to implement our SOA. There are just too many dependencies between the screen programs and the actual RPG application code. You could resort to accessing the database directly to get at information that we need, but this could be dangerous if you try to update database fields that the RPG application does not expect to be updated. Even for reading information this strategy has its drawbacks since in many cases you are circumventing or duplicating the security logic that is found in the RPG application, and if we are using a packaged solution we can never be guaranteed that the database will not change from release to release.

Application Architecture Modernization is the Key

Modernization of legacy application architecture is the key if your ultimate goal is to provide a rich user interface and a services oriented architecture which encapsulates the business processes that can be re-used by multiple applications. Modernization implies that the RPG code will have good separation of concerns with user interface, business logic and data in separate layers, and that the application will be more modular so that the various parts of the business logic are easier to access.

No Big Bang

One of the corner stones of SOA is never to try the big bang approach to creating your SOA layer. You need to carefully assess the business needs and identify which business processes are used over and over again by multiple applications. In this way you can prioritize the interfaces/services that will provide the highest value to your organization. For example, an organization may decide that their order-to-cash process is important because it is one that many applications including: eCommerce; CRM; and Call Center, need to access. You would create an SOA project to provide a generic set of business services that encapsulates your order-to-cash process. A big part of this project would be to modernize their ERP systems order-to-cash programs so that it could be used by the SOA layer.

Use a Tools Based Approach

Do not do modernization “by hand”; you cannot afford the cost and the frustration! You should use a tools based approach. There is no magic tool that will do a 100% conversion of legacy code to a modern architecture. However, there are tools that do a pretty good job and will get you 70-90% there, depending on the complexity of your code. One of the most mature tools for iSeries modernization is *Databorough's X-ANALYSIS*. This tool will help you understand your RPG application and will also help you restructure your application into a more modern layered architecture with separation of user interface, business logic, and business data X-Migrate. In fact the toolset from Databorough will also provide a way of creating an AJAX UI (based on Google's GWT framework), and will provide a Web Services interface to the RPG business logic(X-Modernize), that can then be used in your SOA layer.

Ensure you have single code base

Once you modernize your application architecture you should only have a single code base. Do not try to maintain multiple code bases for the same business process. Even if you are modernizing your user interface, you should ensure that your 5250 based UI, and your modern AJAX based UI go through the same code base. Ideally this would also be through the SOA layer, but it may not be practical to change the 5250 display programs to go through the SOA layer.

If you are using a packaged solution, and your application vendor claims to have modernized their user interface and their application architecture, make sure that you ask them if the modern UI and the 5250 UI go through the same code base to drive the business logic. If they do, then the vendor has modernized the application in such a way that you will be able to easily customize and maintain it. If not, then you will be customizing and maintaining multiple code bases which will be much more costly and error prone, and which you should avoid.

Future Proofing

The phrase “future proofing” is used to describe the difficult process of trying to anticipate future changes and taking action to minimize the negative impact on applications you are currently developing. This is an important concept in general for applications since there are bound to be changes in data or business processes, so there must be some degree of future proofing to easily accommodate these changes.

Whenever you use new technologies like AJAX or SOA that are in the early stages of their “technology lifecycle” and have immature standards, you have to consider what the effects on your applications will be in the future as the technology matures. With UI technology in general the only thing that you can be certain of is that it will change.

One of the best mechanisms for future proofing is to use application generators. That is, a tool, or set of tools that allow you to describe your application, and then the tool generates the ultimate code. This provides some level of future proofing in that the tool can generate new UI technologies as the UI technology changes. Some tools even generate to multiple UI technologies from the same application description. This allows you to deploy your application in multiple environments. The Databorough toolset mentioned above is based on code generator technology, so to a certain extent it will provide a high degree of future proofing for your application.

Conclusions

When you are assessing any new technology you should always assess them based on the business problems your organization is trying to solve, and where they are in their technology life cycle. AJAX and SOA are two exciting new technology trends in the early stages of their technology life cycle that industry analysts like *Gartner Group* and *Forester Research* have forecasted to potentially have a “high impact” on enterprises. AJAX brings the promise of client/server type rich user interfaces to the web, and SOA brings the promise of the creation of shared re-useable services that can be used by multiple applications, or used to create composite applications. These technologies go

hand-in-hand in the creation of applications with a compelling web-based rich user interface. However, the key to unlocking the promise of both of these technologies lies in leveraging of legacy business logic and data. One of the great inhibitors in properly leveraging our legacy iSeries application resources is lack of a modern architecture with separation of concerns. Ultimately you will need to modernize your legacy iSeries application architecture, but you should use a phased approach and leverage modernization tools to help you with the effort. Once completed, your organization will be in a position to use your application assets in a more agile and dynamic fashion to solve your business problems.

For more information on the tools and methods described in this paper, please contact Stuart Milligan at:

stuartm@databorough.com

705 719 7952