
Enterprise AJAX

By David S. Linthicum

Enterprise AJAX

What do AJAX and service-oriented-architecture SOA have in common? The answer: Everything.

Is AJAX an enterprise technology? The answer: Absolutely.

As we move to next-generation enterprise architectures using newer notions such as SOA, there's a need for a dynamic Web interface that can layer over services and provide more value to the enterprise. Moreover, the enterprise in general can benefit from the advantages of AJAX; it's just a matter of making enterprise developers and SOA architects aware of AJAX.

AJAX is becoming the standard dynamic interface for the Web. It adds value to SOA as well, providing the core-enabling technology for user interaction no matter whether we're dealing with applications that are remotely hosted or local to the enterprise.

In essence, AJAX provides better edge technology for SOAs, or the top layer of technology dealing with the user interface. AJAX can extend visual service to a true interactive dynamic interface that's more attractive and functional for the end user.

The benefits of AJAX to the enterprise are clear and include:

- **The ability to leverage the same interface technology whether you're dealing with local or remote sites or applications.** What's key about AJAX is that many enterprises can agree that it's the standard interface technology and, as such, standardize on it as a common platform-agnostic user interface. It doesn't matter if the AJAX interface is delivered on Windows, Linux, or the Mac. This makes deploying service-oriented enterprise applications that much easier, avoiding platform localization and testing issues.
- **The ability to leverage Web Services using a more dynamic and rich interface than traditional browser technology.** While a browser is functional for Web-based applications, the lack of interactive and dynamic behavior limits its use in the enterprise. AJAX doesn't use the same "pump and pull" model that traditional HTTP-driven browser-based applications leverage. AJAX provides native-like application inter-

faces and performance, functioning as good as or better than native interface APIs, such as Win32.

- **The ability to create mashups to solve specific business problems quickly using standard dynamic interfaces that front services.** Mashups are powerful ways of taking existing applications and services and creating something even more useful. AJAX provides better enabling technology to facilitate creating mashups and combining dynamic applications into a single interface with additional binding logic. Using this paradigm, enterprises can quickly create such useful mashups as integrating Google Maps with their delivery system.

The Emergence of the Rich Client for SOA and the Enterprise

Considering the architectural discussion above, as we look to make more practical use of Web Services, the need has emerged for a better user interface: one that's neither too fat nor too thin. We need an interface that lets developers make the most of the client's native features, while, at the same time, not bogging the client down with services that are better kept at the back end. We call this new hybrid interface a rich client and AJAX is an instance of rich client-enabling technology.

However, let's back up a bit. A rich client is a small piece of software that runs on the client to leverage and aggregate back-end Web Services, letting them appear like a single, unified native application. Indeed, a new interface is needed as both developers and end users begin to understand the limitations of traditional Web-based interfaces that are the current interface-of-choice for many distributed applications. Figure 11.1, for instance, is a rich client interface embedded in Salesforce.com for application integration services. Notice how it supports drag-and-drop and the click-and-drag interface process. Impossible with traditional HTTP approaches to application development.

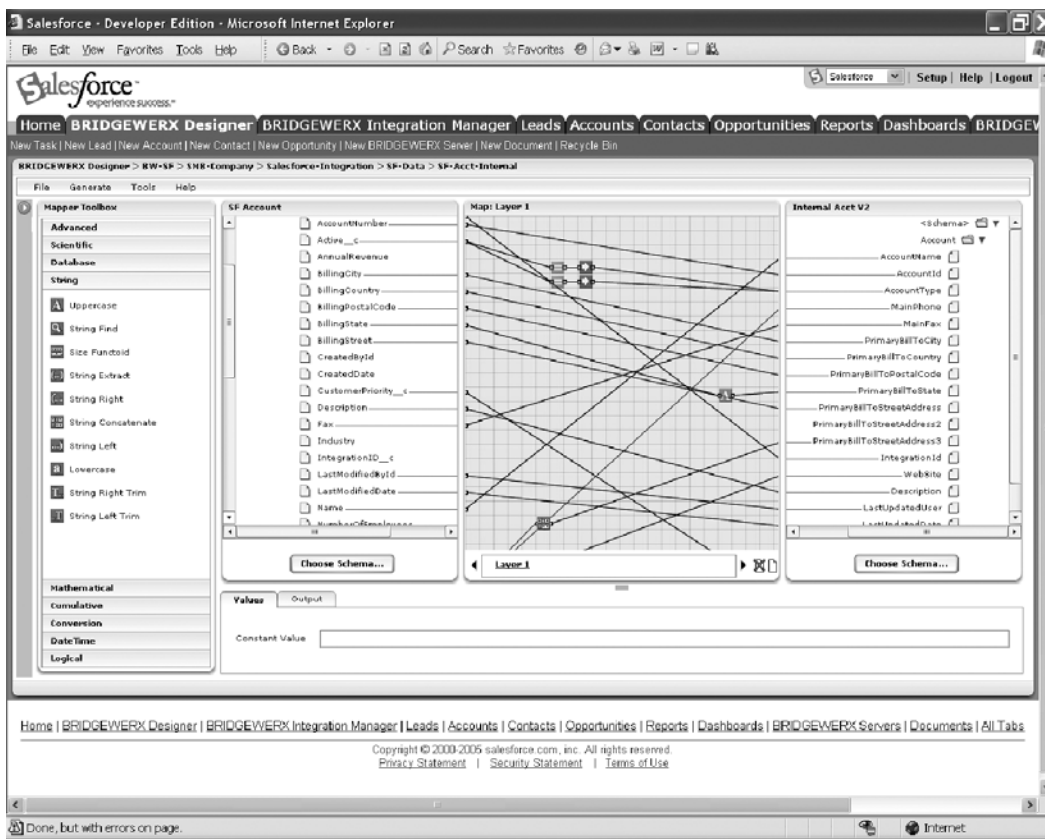


Figure 11.1

Why a rich client when deploying interfaces in enterprises? Truth be told, Web interfaces, widely used in enterprises, were never really designed to support true interactive applications. The Web was built as a content provider, serving up documents and not dynamic application services. If you think about it, you're reloading document after document to simulate an interactive application and always have to go to the back-end Web server to ask for new content. Very little occurs at the client.

As the Web became popular and we looked to support business applications in the enterprise using the Web interface, we began to create new mechanisms to deliver dynamic content including dynamic HTTP/HTML pushers (e.g., CGI, ASAPI, and ISAPI) and new browsers that supported complex dynamic behavior. We're at such an advanced state today that entire enterprises run most of their relevant business applications using Web interfaces.

However, with the advent of Web Services and SOA, and the need to leverage dynamic behavior within the interfaces, traditional browsers fall way short. Their get/push model for driving inter-

faces isn't well suited to SOAs, which are, in essence, remote functions that are better for more visually rich types of interfaces, such as the more traditional GUI client/server interfaces popular a few years ago.

Rich clients are not a revolution, but an evolution of technology, including AJAX. Today we look to leverage dynamic behavior and deliver that experience directly to the end user, aggregating Web Services in an interface that appears as much like a native application as possible.

As said above, rich clients employing AJAX provide capabilities that thin clients can provide, including windowing features and data navigation control such as buttons, checkboxes, radio buttons, toggles, and palettes. They can also integrate content, communications, and platform-independent application interfaces for distribution through emerging SOAs. The rich client using AJAX becomes a Web Services/SOA terminal of sorts, letting applications communicate and even execute on one another in a distributed environment.

This is great news for those who are developing Web Services or implementing an SOA. With rich clients, suddenly those services have a much higher value. Indeed, you can mix-and-match services in a rich client to create some very valuable applications. Perhaps, someday, the use of static and dynamic HTML and heavyweight protocols such as HTTP won't be the primary way we view distributed applications. Rich clients let us view applications that look and act like native client programs, even running remotely. That is a step in the right direction and the reason AJAX is so important to SOA.

So What's an SOA and Where Does AJAX Fit?

SOAs are like snowflakes...no two are alike. Moreover, everyone has their own definition of an SOA including everything from messaging systems to portals. However, many common patterns are beginning to emerge.

First, let me put forth my definition of SOA so we're working from the same foundation before we figure out where AJAX fits.

To me an SOA is a strategic framework of technology that lets all interested systems, inside and outside an organization, expose and access well-defined services, and information bound to those services, that may be further abstracted to orchestration layers, composite applications, and interfaces for solution development.

Pay special attention to the interfaces part.

Why do we build SOAs? The primary benefits of an SOA include:

1. Reuse of services/behaviors or the ability to leverage application behavior from application-to-application without a significant amount of re-coding or integration. In other words, the ability to use the same application functionality (behavior) over and over again without hav-

ing to port the code. Leveraging remote application behavior as if it existed locally.

2. Agility or the ability to change business processes on top of existing services and information flows quickly and as needed to support a changing business. Overall, the consensus is that agility is more valuable than reuse.
3. Monitoring or the ability to monitor points of information and points of service in real-time to determine the well-being of an enterprise or trading community. Moreover, the ability to change or adjust processes for the benefit of the organization in real-time.
4. Extend the reach or the ability to expose certain enterprise processes to other external entities for the purpose of inter-enterprise collaboration or shared processes. This is, in essence, next-generation supply chain integration.
5. The ability to put dynamic interfaces on both abstract data and services, letting the architect place volatility in a single domain between the services and the interface. This is where AJAX adds a tremendous amount of value.

The notion of an SOA isn't at all that new. Attempts to share common processes, information, and services have a long history, one that began more than 10 years ago with multi-tier client/server — a set of shared services on a common server that provided the enterprise with the infrastructure for reuse and now provides for integration — and distributed object movement. Reusability is a valuable objective. In the case of an SOA, it's the reuse of services and information bound to those services. A common set of services among enterprise applications invites reusability and, as a result, significantly reduces the need for redundant application services.

What's unique about an SOA is that it's as much a strategy as a set of technologies, and it's really more of a journey than a destination. Moreover, it's a notion that's dependent on specific technologies or standards such as Web Services and interface technology such as AJAX but really requires many different kinds of technologies and standards for a complete SOA. The kinds of technologies you employ are dependent on your requirements. As mentioned above, all SOAs are a bit different; sometimes very different.

Let's be a bit clearer as to where AJAX fits in this SOA mix by providing core reference architecture or the basics of SOA. Figure 11.2 is a diagram of the SOA logical architecture, working from the most primitive to the most sophisticated, top to bottom.

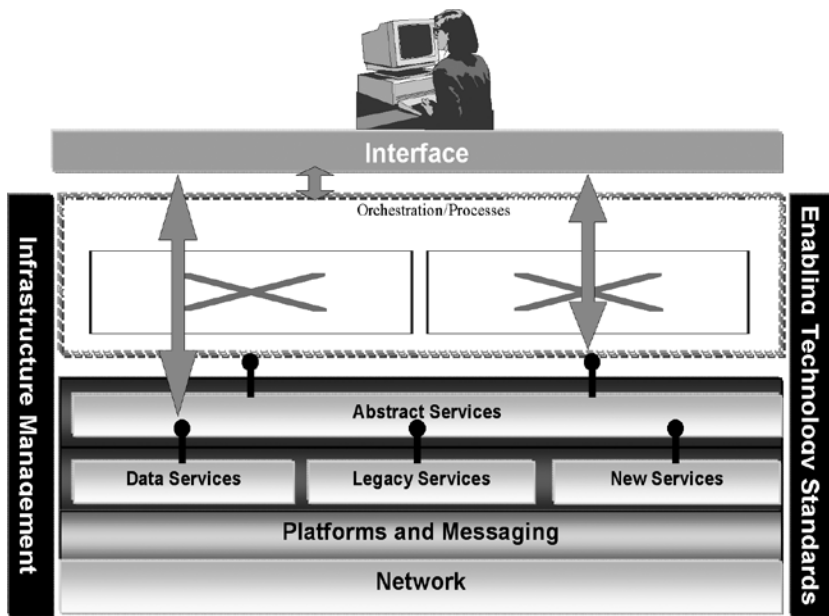


Figure 11.2

Base Services

At the lowest level you have base services, including legacy services, new services, and data services.

Legacy services, such as existing mainframe or ERP systems, can expose services typically through proprietary interfaces such as LU6.2 ACCP, or SAP's BAPI. These services usually provide both behavior and information bound to that behavior. In other words, there is functionality and structure.

New services are those services created from the ground up as services. These services have behavior as well as information bound to the behavior, but are built from scratch as services, so there's not much further abstraction required (see next level up). They are typically Web Services but don't have to be as a rule.

Data services, as the name implies, are databases, data files, or other data stores that can produce and consume data. They support some behavior, but just enough to manage the data interaction services.

Abstract Services

Abstract Services are services that exist on top of base services, in essence, putting easier-to-use and better organized layers on legacy, new, and data services. It's the role of the abstract layer to create order out of the base services that are typically raw services from existing systems and data

sources. This layer of abstraction provides the following features and benefits:

1. A mechanism to normalize both services and data so they are managed better by the upper layers
2. A way to filter out services that are irrelevant to the SOA
3. An easier approach to management and governance

Orchestration

For our purposes we can define orchestration as a standards-based mechanism that defines how Web Services work together. In this case, we're talking about the abstract service at the lower layer, including business logic, sequencing, exception handling, and process decomposition, such as service and process reuse.

Orchestrations can span a few internal systems, systems between organizations, or both. Moreover, orchestrations are long-running, multi-step transactions, almost always controlled by one business party, and are loosely coupled and asynchronous in nature.

We can consider orchestration as really another complete layer over abstract services per our architecture. Orchestration encapsulates these integration points, binding them together to form higher-level processes and composite services. Orchestrations should actually become services.

Orchestration is a necessity if you're building an SOA, intra- or inter-organization. It's the layer that creates business solutions from the vast array of abstract services and from information flows found in new and existing systems. Orchestration is a god-like control mechanism that can put our SOA to work, as well as provide a point of control. Orchestration layers let you change the way your business functions to define or redefine any business process on-the-fly as needed. This provides the business with the flexibility and agility needed to compete.

Orchestration must provide dynamic, flexible, and adaptable mechanisms to meet the changing needs of the domain. This is done by separating the process logic and the abstract services used. The loosely coupled nature of orchestration is key since all services don't have to be up and running at the same time for orchestrations to run. This is also essential for long-running transactions. And, as services change over time, there's usually no need to alter the orchestration layer to accommodate the changes. At least, not if they're architected properly.

Orchestration has the following properties:

- A single instance of orchestration typically spans many instances of services and even organizations.
- Most orchestrations leverage public standards such as BPEL.
- Orchestrations can be public – available to everyone – or private – available just to the owner – or shared – for supply chain integration scenarios.
- Orchestrations are usually driven from a single party; they're not always collaborative.

- Orchestrations themselves can become services that are available to other services or orchestrations.
- Orchestration is independent of the source and target systems. Changes can be made to orchestration without having to force changes to the source or target systems. In other words, this architecture is loosely coupled.
- Orchestrations are always decomposable down to the base processes in the source or target systems.

Interface

The interface layer is where AJAX lives. The purpose of the interface layer is to make services – core, abstract, or those exposed through orchestration (see Figure 11.2) – available to human beings. In this architecture, AJAX communicates directly with these services through its asynchronous mechanisms and exposes the information or behavior to the user.

In the interface layers, SOA developers can mix and match services and information and bind them to a dynamic interface in a way that makes sense for the end user. For instance, you can take an abstracted data service to populate a customer list and a risk service to process against that list and another abstract data service to put the information back in the data store (see Figure 11.3).

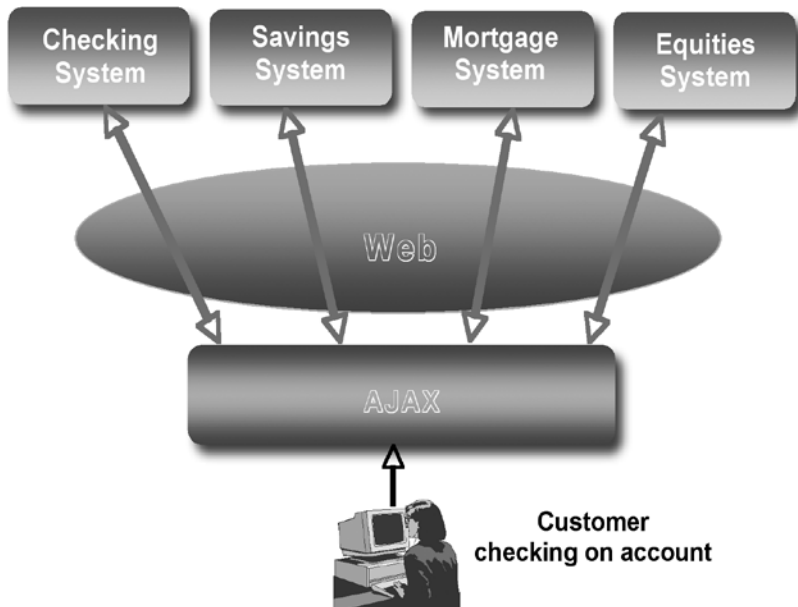


Figure 11.3

By the way, this mechanism is the same with other interface development technologies, including Java, C++, and Ruby on Rails. However, AJAX's use of a more asynchronous interface makes it better

suited to this type of application with an SOA and, as such, applies anytime you're interacting with abstract, orchestrated, or core services.

It's also helpful to note that the interface layer can interact with any service at any layer, including the core, abstract, and orchestrated services, and should be able to interact with services that are either course- or fine-grained.

Understanding SOA Levels

All SOAs aren't the same. As we deploy SOAs, I see patterns beginning to emerge that range from very primitive to very sophisticated, from low value to high value. The question is: What level is your SOA?

Level zero SOAs are those that simply send SOAP messages from system to system. There's little notion of true services. Instead they leverage Web Services as an information integration mechanism. Hardly a SOA but certainly a first step.

It's also important to note that you don't need Web Services to create an SOA. This is true for all levels.

Level 1 SOAs are those that also leverage everything in Level 0, but add the notion of a messaging/queuing system. Most Enterprise Service Buses (ESBs) are Level 1 SOAs, leveraging a messaging environment that uses service interfaces, but don't really deal with true services (behavior), and instead move information between entities like messages through queues.

While services are a part of Level 1 SOAs, they're really all about information and not about application behavior. For instance, while you invoke a service to push a message into a queue and retrieve a message off a queue, it's really leveraging services as a well-defined interface and not accessing application functionality. Sometimes SOA architects attempt to abstract application behavior using an ESB. If that's the case, you're moving up to a Level 4 SOA. However, doing this is typically more trouble than it's worth because you're dealing with information-oriented integration technology that's attempting to deal with services/behavior – an unnatural act.

Level 2 SOAs are those that leverage everything in Level 1 and add an element of transformation and routing. That means that the SOA can not only move information from source and target systems, leveraging service interfaces, but can transform the data/schemas to account for the differences in application semantics. And by adding the element of intelligent routing, you can route the information based on elements such as source, content, and logical operators in the SOA.

Level 3 SOAs are those that leverage everything in Level 2, adding a common directory service. The directory provides a point of discovery of processes, services, schemas, and such, letting all those leveraging the SOA easily locate and leverage assets. Without directories,

the notion of service reuse – the real point of building an SOA – won't work. Directories are typically standards-based, including UDDI, LDAP, and sometimes more proprietary directories such as Active Directory.

Level 4 SOAs are those that leverage everything in Level 3, adding the notion of brokering and managing true services. Here's where brokering of application behavior comes into play. In other words, at this level, we're not only talking about managing information movement, but discovering and leveraging true services.

At this level we can broker services between systems, letting the systems both discover and leverage application behavior as though the functionality was local. This is the real goal of Web Services – the ability to share services and not worry about platform-specific issues or where the services are actually running.

What's important here is that we understand that the value is in the behavior, as well as the information bound to that behavior. This level of an SOA can provide for discovery, access, and management. Most SOAs are built with Level 4 capabilities in mind, but may work up to them from the lower levels. If you do that, make sure you're leveraging the right technology and standards that support all levels.

Finally, Level 5 SOAs are those that leverage everything in Level 4, adding the notion of orchestration. Orchestration is key, providing the architect with the ability to leverage exposed services and information flows, creating, in essence, a “meta-application” above the existing processes and services to solve business problems.

Actually, orchestration is another complete layer up the stack, over and above more traditional application integration approaches we deal with at the lower levels. Thus, orchestration is the science and mechanism of managing the movement of information and the invocation of services in the correct and proper order to support the management and execution of common processes that exist in and between organizations and internal applications. Orchestration provides another layer of easily defined and centrally managed processes that exist on top of existing processes, application services, and the data in any set of applications.

The goal of this kind of SOA is to define a mechanism to bind relevant processes that exist between internal and external systems to support the flow of information and logic between them, maximizing their mutual value. Moreover, we're looking to define a common, agreed-on process that exists between many organizations and has visibility into any number of integrated systems, as well as being visible to any system that needs to leverage the common process model.

As services – and the architectures that support them – become more of an asset to the enterprise, we need to begin to learn how to categorize the architectural patterns. Hence, the SOA levels discussion. This provides both a better understanding of what a true SOA is and lets us pick the right level to meet our business needs.

Enterprise AJAX Tools

While AJAX is a relatively new notion to the Web, and the enterprise, there are a few enterprise tools that are beginning to appear. The most promising are Microsoft's Atlas and Tibco's General Interface. Let's take a quick look at each one.

Microsoft Atlas is a new Web development technology that integrates client script libraries with the ASP.NET 2.0 server-based development framework. Atlas offers the same kind of development platform for client-based Web pages that ASP.NET offers for server-based pages (see Figure 11.4).

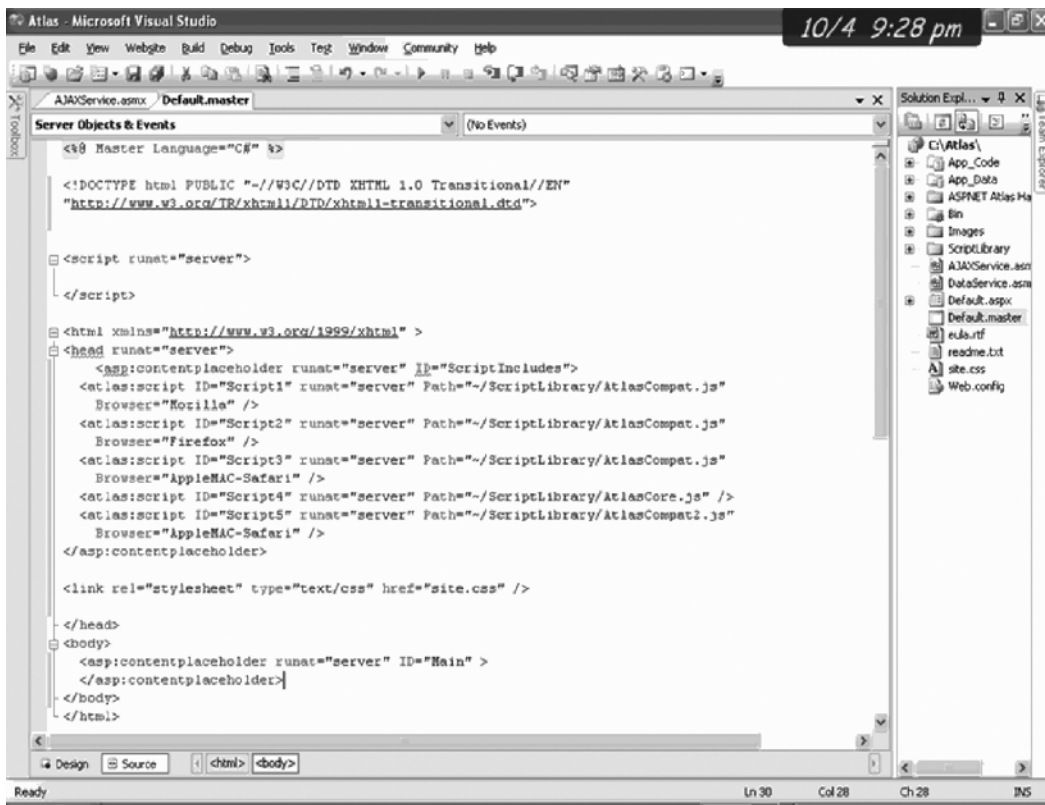


Figure 11.4

Atlas makes it possible to take advantage of AJAX techniques on the Web and enables you to create ASP.NET pages with a rich client and server communication. Atlas isn't just for ASP.NET. You can take advantage of the rich client framework to build client-centric Web apps that integrate with any back-end data provider, including data services.

TIBCO General Interface provides developers with drag-and-drop visual authoring tools for standard XML and XSD, SOAP, and WSDL communications, as well as HTTP/S GET and POST opera-

tions. General Interface users have access to TIBCO's General Interface Developer Community, an online resource. Also, by using its add-in architecture, TIBCO General Interface lets additional third-party packages plug into TIBCO General Interface (see Figure 11.5).

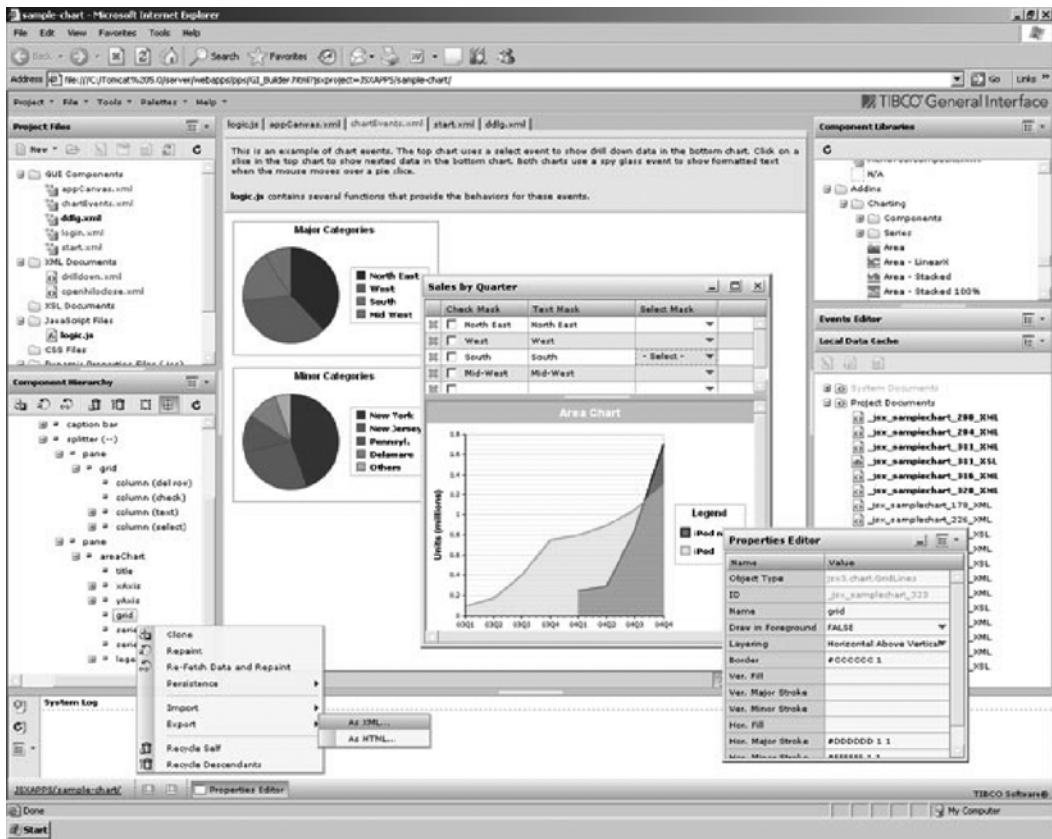


Figure 11.5

General Interface works with existing browser capabilities, letting users get a full-featured application instantly from a URL. The product claims to reduce development time and cost by using familiar APIs, visual authoring tools, step-through debugging, and extensible, reusable components. The visual tools are a Web-based application powered by General Interface using AJAX in Internet Explorer.

Summary

AJAX is a mere instance of a rich client interface for both SOA and the enterprise. It's the momentum behind AJAX that will ensure its place in most enterprises looking to employ rich clients, which are most enterprise-class businesses. However, this technology isn't always a slam-dunk.

CHAPTER 11

You must first address your requirements before leveraging AJAX or, for that matter, any other technology.

At the end of the day, AJAX is just another part of the SOA solution and it needs to exist with other robust technologies that solve the problems at hand. Therefore, you must consider using AJAX holistically and in the context of other enabling technologies, standards, and the ultimate architecture.

Unlike traditional application development, where the database and application are designed, SOAs are as unique as snowflakes. When all is said and done, no two will be alike. However, as time goes on, common patterns emerge that let us share best practices when creating an SOA. We still need to travel further down the road before we can see the whole picture.

