THE ARCHITECTURE JOURNAL™
Input for Better Outcomes
Journal 12

Web Architecture

Web 2.0 in the Enterprise

Efficient Software Delivery through Service Delivery Platforms

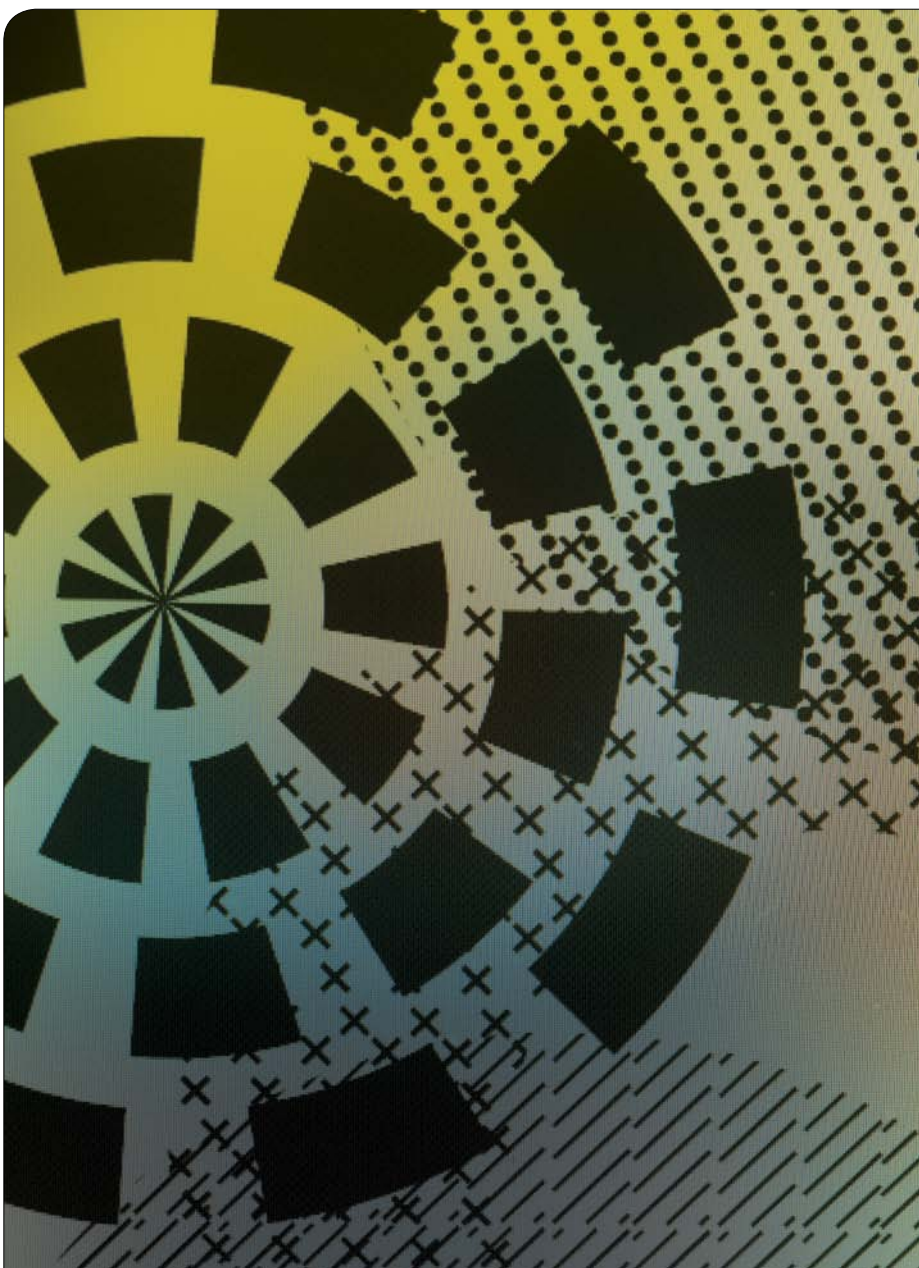
Secure Cross-Domain Communication in the Browser

An Application-Oriented Model for Relational Data

Architecture Journal Profile: Pat Helland

Data Confidence Over the Web

Managed Add-Ins: Advanced Versioning and Reliable Hosting

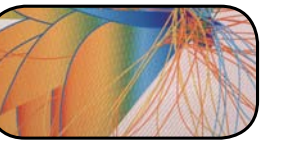


Microsoft®

Contents

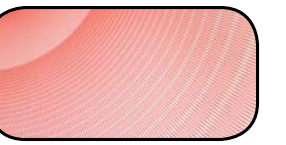
THE ARCHITECTURE JOURNAL™
Input for Better Outcomes
Journal 12

Foreword 1
by Simon Guest

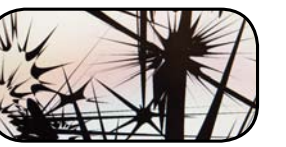


Web 2.0 in the Enterprise 2
by Michael Platt
Discover the technology, data, and people elements of a Web 2.0 architecture and how they can be used inside and outside the enterprise.

Efficient Software Delivery through Service Delivery Platforms 7
by Gianpaolo Carraro, Fred Chong, and Eugenio Pace
Explore the use of a Service Delivery Platform to enable efficient software delivery for the Web.



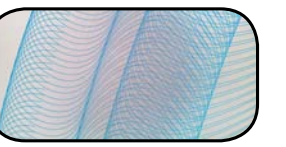
Secure Cross-Domain Communication in the Browser 14
by Danny Thorpe
Today's browsers work inefficiently across multiple domains. Learn a new technique for developing Web sites that can share information.



An Application-Oriented Model for Relational Data 19
by Michael Pizzo
Often the shape of data in a database does not match the application that interacts with it. See how a conceptual model can be used to overcome this.

Architectural Journal Profile: Pat Helland 26
Pat Helland is an Architect in Microsoft's Visual Studio team. Get the update on his career and thoughts on becoming an Architect.

Data Confidence Over the Web 28
by Peter Hammond
Many patterns exist for the exchange of data over the Web. Learn recommendations and pitfalls to avoid from a series of experiences.



Managed Add-Ins: Advanced Versioning and Reliable Hosting 33
by Jesse Kaplan
Learn about some of the advanced architectural concepts in the next release of the .NET Framework that support a resilient and reliable add-in model.

Foreword

Dear Architect,

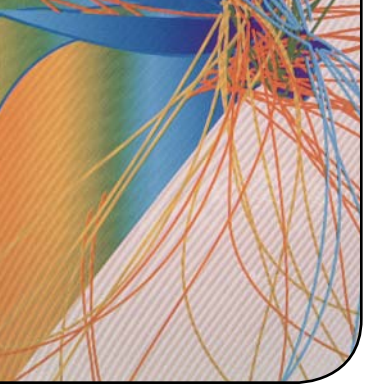
Welcome to Issue 12 of the *Journal*. The theme this time is "Web Architecture." I'm sure I don't need to tell you how important the Web has become in recent years for architects in almost all organizations. After speaking with many about designing for the Web, however, one common motif is the need to adapt, to take architectural principles that have worked in the past and refactor them for the Web. As a result, I've tried to keep this in mind when selecting the articles for this issue.

Leading the issue with the topic of Web 2.0 in the Enterprise, we have returning author, Michael Platt. Michael discusses some of the technology, data, and people principles of Web 2.0 and then maps them to how they can be used inside and outside the enterprise. Following Michael's article, Gianpaolo Carraro, Fred Chong, and Eugenio Pace introduce the concept of a Service Delivery Platform. Building on their work in the Software as a Service space, they cover what's needed to enable efficient software delivery on the Web today. Danny Thorpe from the Windows Live team follows with an article that explores some of the complexities of cross-domain communication in the browser. Using a novel analogy, Danny looks at a set of techniques that can be used to overcome this challenge. We also have Michael Pizzo's excellent article on an application-oriented model for relational data. With data being a huge part of many applications on the Web, Michael takes us through some of the challenges with database and application schemas today, and introduces a framework to better integrate the two.

In my introduction for the last issue, I mentioned a former colleague of mine, Pat Helland. For this issue, I caught up with Pat, who has recently returned to Microsoft as an Architect in the Visual Studio team. I ask Pat about the role that he is going to be playing, and some of his colorful thoughts on how to become an architect. Following Pat's interview, we have an article from Peter Hammond. Peter takes us on a journey over the past 10 years, looking at some data issues that many people will be able to relate to, and introducing a technique for reliable exchange of data over the Web today. To wrap up this issue, we are lucky to have an article from Jesse Kaplan. Jesse takes us through some of the architectural considerations in the next release of the .NET Framework, and introduces a model for creating reliable and resilient add-ins.

Well, that wraps up another issue. Talking of adapting, here at *The Architecture Journal* we are always trying new things—a lot based on what we hear from you. If you have feedback or ideas for upcoming themes and articles, we'd love to hear them in order to make this a better publication for you. You can reach us at editors@architecturejournal.net.

Simon Guest



Web 2.0 in the Enterprise

by Michael Platt

Summary

The old models of how people publish and consume information on the Web have been radically transformed in recent times. Instead of simply viewing information on static Web pages, users now publish their own content through blogs and wikis and on photo and video sharing sites. People are collaborating, discussing and forming online communities, and combining data, content, and services from multiple sources to create personalized experiences and applications.

Commonly and collectively called "Web 2.0," these new content sharing sites, discussion and collaboration areas, and application design patterns or "mashups" are transforming the consumer Web. They also represent a significant opportunity for organizations to build new social and Web-based collaboration, productivity, and business systems, and to improve cost and revenue returns.

This article will examine how technology, data and people come together to make up Web 2.0; how Web 2.0 is being used in the consumer space today; and how these techniques and concepts can be used both inside and outside the enterprise to provide new productivity and business opportunities.

Web 2.0

Web applications have undergone significant change over the last decade; ten years ago, there were no Web sharing sites or applications, merely sites composed of static pages or ecommerce applications. Companies that had customer-facing Web sites were able to connect with Internet-savvy consumers and use their Web sites as a channels to market and sell their products; corporate intranets were used mainly as places to post news and company policies. More recently, Web sites have become destinations for communities of users to create and share rich and complex data, such as music, images, and video, and to discuss and rate that content.

This phenomenon was dubbed "Web 2.0" in a seminal discussion paper by Tim O'Reilly in September 2005, and it continues expanding today. In essence, Web 2.0 is the collective realization that the ability to use the Web to write as well as read rich content—combined with support for social networking and the rapid spread of broadband access—allows people to interact with the Web, with online content, and with one another. It represents a fundamental change in the way people interact with content, applications, and other users—the new Web is a platform for harnessing and promoting collective intelligence. People are no longer merely consumers of content and applications: They are participants, creating content and interacting with different services and people. More and more people are creating blogs, contributing to knowledge bases such as Wikipedia, and using peer-to-peer (P2P) technologies. Sometimes referred to as the network effect, this increase in participation and content creation presents new opportunities to involve the user in deeper, more meaningful ways.

Tim O'Reilly originally defined the characteristics of Web 2.0:

- The Web as a platform
- Harnessing collective intelligence
- Data as the next Intel inside
- End of the software release cycle
- Lightweight programming models
- Software above the level of a single device
- Rich user experience.

These can be grouped into three areas: the use of the Web as a platform, the Web as a place to read and write rich content, and the social and collaborative use of the Web.

The Web as a Platform

Web 2.0 systems use the Web as a platform, as a huge range of interconnected devices which can provide a new level of rich and immersive experience for the user, an easy-to-use and lightweight programming model for the developer, and a rapid and flexible deployment mechanism for the supplier. Web 2.0 reenvisions the Internet from the user, the developer, and the supplier's perspectives, each of which allow new and creative uses of the Internet.

The concept of service underpins all connected systems, including, of course, Web 2.0. A service-based system is based on the principle of separation of concerns through the use of loose coupling or message passing. Loose coupling allows functionality to be created as a service and delivered over a network; for example, in the Web 2.0 world, diary functionality can be provided by a blog engine and be delivered as a service to the end user or blogger over the Internet.

This delivery of software functionality over the Internet is commonly called Software as a Service (SaaS) and underpins most Web 2.0 systems today.

When we consider the Internet as a platform, we can see that it has to provide a number of important platform elements such as device independence, rich and common user interface, a common programming interface, and a software or service deployment and management mechanism.

Software above the level of a single device

We are very familiar with software on a server providing services to software on a PC (in Windows or in a browser), which then consumes or displays them. While this is a well-understood model, it does not cover a number of common cases, such as P2P systems or delivery to non-PC devices such as music players, phones or navigation devices. We need a model which includes these cases and understands a higher level of service than basic HTTP to connect them—it needs to address the concepts of a music service such as iTunes or Napster or a phone service such as Skype. We need to have a model which addresses software above the level of a single device and a single service but which includes rich, high-level services interconnecting a mesh of different device types in a symmetric manner.

Probably the best example of this type of high level service is Xbox Live where gaming services are supplied between specialist hardware devices working in a peer-to-peer manner. This model is the general-purpose case of service-based computing, and it is what Microsoft is calling a Software + Services model of computing.

Rich user experience

The value of rich and immersive user experience has been well-understood in the PC world since the advent of Windows and has been a focus of browser-based applications for many years, with JavaScript and DHTML being introduced as lightweight ways of providing client-side programmability and enriching user experience in what is commonly called Rich Internet Applications (RIA).

The ability of the Web to provide this RIA functionality was first shown by Outlook Web Access (OWA) which used JavaScript and DHTML to provide full Windows-type interactivity in the browser. The collection of technologies used to provide these rich and dynamic browser-based systems has been called Ajax, standing for Asynchronous JavaScript and XML. Ajax isn't a single technology or even a set of new technologies, but rather several technologies being used together in powerful new ways to provide RIA functionality. Ajax includes:

- standards-based presentation using XHTML and CSS
- dynamic display and interaction using the Document Object Model
- data interchange and manipulation using XML and XSLT
- asynchronous data retrieval using XMLHttpRequest
- JavaScript as the programming metaphor.

Ajax is a key component of most Web 2.0 applications and is providing the ability to create Web applications as rich and dynamic as Windows-based applications—indeed, we are now seeing the advent of Ajax-based applications that can work while disconnected from the Internet and thus provide offline functionality similar to Windows-based clients such as Outlook.

There are also sets of technology other than Ajax which are increasing the value of the user experience in areas such as communications, voice, and video. Instant messaging (IM) is heavily used in Web 2.0 applications to provide instant communications, and there are a wide range of agents and delivery options available for IM systems. Voice over IP (VOIP) systems allow voice and teleconference communication over the Internet as part of the user experience. And finally, the provision of real-time, stored and broadcast video rounds out the client experience.

The breadth, richness, and flexibility provided by these technologies transforms the user interface beyond a dynamic UI to a full interactive audio visual experience, with new, powerful ways for people to interact with systems and one another still being explored.

Lightweight programming models

In Web 2.0, the programming models, concepts and techniques are significantly different from those that have been used in the enterprise. While they are services-based and underpinned with the concept of messaging passing, they use Representational State Transfer (REST) protocols, and focus on simplicity and ease of use.

Web 2.0 programming is based on the concept of separation of concerns using a loosely coupled, message-passing-based model on top of an Internet-based, standard set of communications protocols (HTTP) which is often called RESTful programming. It implies the acts of syndication and composition where services are provided without knowledge of their use. This is very different from a conventional tightly coupled, transactional and object-oriented system. It has a different set of benefits, such as flexibility and speed of implementation, and challenges, such as integrity and management.

The languages—such as Perl, Python and Ruby—and frameworks used in Web 2.0 are simple and dynamic, providing a low barrier to entry and reuse and high productivity. The frameworks have built-in support for common design patterns, such as Model View Controller (MVC), and methodologies, such as agile software development. They are quick and easy to learn, use, and be productive in.

Web 2.0 applications are inherently composable and compositable; because they are built with lightweight programming models and standards-based services, new applications can be created by composing or "mashing up" present applications and services. Mash-ups are where applications and services are composed at the UI; composition is the more general case of services being reused, but both are easily supported in Web 2.0 programming.

End of software release cycles and deployment

The concepts behind Web 2.0 strike a new balance between the control and administrative ease of centralized systems and the flexibility and user empowerment of distributed systems. Web applications are by nature centrally deployed, so central services can manage applications and entire desktops automatically. SaaS builds on this concept to provide the idea of software and services delivery over the Internet. Web 2.0 builds on top of SaaS to provide social and content services over a SaaS mechanism.

This usage of SaaS by Web 2.0 as a deployment and release methodology provides all the well-known SaaS advantages of simple deployment, minimized management and administration and, probably the most important, continual update. Thus, one of the most touted

features of Web 2.0 (and SaaS) is the concept that the system is being constantly updated and enhanced, often in real time in response to user requests. Of course, the issue with this "perpetual beta" that the Web 2.0 community has yet to come to grips with is what happens to downstream or offline applications or services when services or data that they've come to rely on a Web 2.0 application to provide disappear or change.

The Read / Write Web

The second important area of Web 2.0 is the focus on data and content, and in particular the ability for people to create and interact with rich content rather than just consume it. If the original Internet provided read access to data, then Web 2.0 is all about providing read and write access to data from any source. This ability for anyone to create content has caused an explosion of availability of content of all sources (and of all types of quality) and has at the same time created a whole new set of issues around vandalism, intellectual property, and integrity of data.

As the bandwidth available to the end user continuously increases, the richness of the content that can be sent over the Internet increases. The original Internet was all about text; Web 2.0 started with music and images and moved into voice and video. Now TV and movies are the content areas that are being investigated as part of Web 2.0.

While people and organizations have been searching, uploading and downloading all this "explicit" data and content on the Web, they have at the same time being creating a huge amount of implicit data about where they are going and what they are doing. This implicit or attention data of Web 2.0 can be used to predict future behavior or provide new attention-based features. This is how the major search engines work; they monitor the queries that people make and use that data to forecast what people are likely to look for in the future. Of course, the collection, storage, and use of this implicit data raise challenging questions about ownership, privacy, and intellectual property (IP) which still have to be satisfactorily addressed.

Another issue with the huge amount of data on the Web is finding things and navigating ways through it. Search engines use the implicit data (or page ranks) to find textual data but this does not work very well with images or audio data. In addition, in many cases the search engine may not have enough contextual information to provide a valid result. In these cases tagging of the data becomes a valuable way of assisting with data navigation. Web 2.0 applications use tagging and "tag clouds" extensively as a way of finding and navigating through the vast amount of data available on the Web.

Tag data is data about data, or metadata, and one of the major issues with data and content on the Web is that caused by the lack of standards for metadata and schema. It is impossible to cut and paste something as simple as an address on the Web because there is no standard format for addresses. We need to understand the different levels of metadata and have standards for what that metadata is, in order to liberate data on the Web and, in particular, to allow applications to composite data. This standardization of metadata on the Web is what the Microformats organization is trying to provide.

The Social and Collaborative Web

The third key element of Web 2.0 systems is the concept of social networks, community, collaboration and discussion. People naturally want to communicate, share and discuss; this communication is a key part of understanding, learning and creativity. The unique element that Web 2.0 brings is that of social networks and community which are typically enabled by blogs, discussion groups, and wikis. In Web 2.0, the sheer scale and number of people on the Internet creates an "architecture of participation" where the interaction between people creates information and systems that get better the more they are used and the more people who use them. This harnessing of the collective intelligence creates systems which have more and better information than any one person could generate; it provides the "wisdom of the crowds."

There are a number of different types of collaboration that can occur in Web 2.0 systems:

- *Content based.* People gather and collaborate around a piece of news or content, typically in a blog or a spaces-type environment.
- *Group based.* People gather around an idea or interest such as a hobby and discuss it in forums.
- *Project based.* People work together on a common task or project such as a development project, a book, or even something as large as an encyclopedia using wikis.

All three types of collaboration are supported by Web 2.0 systems.

Web 2.0 in the Enterprise

Organizations of all types and sizes from startups to Fortune 100 companies and from all industry verticals have seen the explosive growth on the Web of social and community sites in the consumer space such as MySpace, YouTube and the deluge of Web 2.0 sites. Enterprises have witnessed the moves of major Web players such as Amazon, eBay, Live, Google, and Yahoo to include social and community elements, and the interest and demand that this has created. They are now actively investigating and in many cases building new community-based portals and businesses for their own organizations; Web 2.0 is moving into the enterprise.

Organizations are interested in using Web 2.0 techniques primarily in two areas: inside the organization to improve efficiency and productivity, and from the organization to the customers to improve revenue and customer satisfaction. The use of Web 2.0 within organizations is called Enterprise 2.0 and is likely to be the first area where Web 2.0 will be used by organizations. The use of Web 2.0 by enterprises to face their customers and consumers is similar to Business to Customer (B2C) activity but with a social and community focus so it is called Business to Community or B2C 2.0. Interest in this use of the "community as a customer" is rapidly growing.

Enterprise 2.0

Enterprise 2.0 or Web 2.0 in the Enterprise is a term coined by Professor MacAfee of the Harvard Business School in 2006 to describe the use of Web 2.0 techniques within an organization to improve productivity and efficiency.

Through the adoption of Web 2.0 techniques, information workers, like their consumer counterparts, are able to control their own user experiences with less guidance from IT, and thus create for themselves a more intuitive and efficient work environment. The end result is improved worker productivity, morale, and satisfaction. Properly understood and deployed, Web 2.0 technologies, methods, and

patterns can be used in the enterprise to great effect, boosting overall organizational productivity and efficiency. In this section, we look at some the elements of Web 2.0 that make this possible.

Rich user experience

Providing users with a single user experience for all their needs increases productivity, minimizes training costs, and encourages deeper adoption and usage. This includes access to information and applications whether they are connected or disconnected, whether they are using a mobile device or a laptop, or whether they are using a thin client or a smart client. The rich user experience support provided by Ajax and graphics subsystems such as Silverlight enable the rich user experience people expect in today's systems.

Lightweight programming models

One of the tenets of Web 2.0 is the concept of user-created applications or "mashups"—of user-generated applications using lightweight programming models. These promise to provide a dramatic change from the bottlenecks and constraints associated with IT-generated applications. With Web 2.0 techniques, users can easily create applications specific to their own needs. However, many current composite application strategies fall short by focusing on simply resurfacing applications in an Ajax interface (UI) and failing to provide true user enablement. To capitalize on the user-driven application opportunity, enterprises must provide departments and users with a managed environment and familiar tools that allow them to easily customize or create their own workspaces and solutions.

End of software release cycles and deployment

The underlying use of the Internet as a platform in Web 2.0 allows the simple, rapid, and flexible distribution of applications and data throughout the organization, removing the user from fixed and inflexible IT upgrade cycles and allowing a new level of organizational support and responsiveness for the user.

Read / write Web

Data and documents are critical to any organization, and service-based systems such as Web 2.0 allow data and document creation, modification and exchange with greatly reduced complexity and enhanced ease of use. The provision of data, content management, and collaboration systems which can support the rich content format and social techniques of Web 2.0 is critical for the use of the read/write web in the Enterprise.

Collaborative Web

Enterprises with large employee, partner, and customer bases have long known the value of the knowledge living in employees' heads and in the databases and unstructured documents found across the organization. Attempts to collect this information into knowledge management systems have been made in the past, with varying degrees of success but Web 2.0 technologies such as blogs, wikis, and enterprise search for people and data may lower the bar to knowledge management and provide a new platform for collaborating on complex and creative tasks. It should, however, be noted that the real barrier to knowledge management is around social and value issues in organizations rather than technical ones—these barriers are not addressed by the Web 2.0 technologies, so it is not clear whether Web 2.0 will successfully enable knowledge management in organizations.

In conclusion, many Web 2.0 techniques can be used in organizations in such areas as rapid application development using mashups and blog- and wiki-based knowledge management.

Business to Community (B2C 2.0)

The area of Web 2.0 usage in the enterprise that has the greatest potential impact from an organizational and revenue perspective is in the customer-facing areas of organizations; it is possible that the whole customer contact, sales, and CRM cycle will be changed by the use of Web 2.0 techniques. In marketing, the opportunity to provide rich, interactive media and close customer interactivity through wikis and blogs will provide new ways of contacting and engaging potential customers. In sales, the use of new form factor devices such as mobile phones to interact with the customer throughout the sales process is another major area for new development. In customer support, the use of experts in the community to assist with problem resolution through discussion groups creates whole new support models. The reasons for this interest are:

- *Revenue and growth.* New revenue streams can be built and present revenue streams enhanced through community and social networking. In particular, the cost containment of the last five years has given way to business-side interest in innovation-based growth and revenue. The rapid growth and innovation in the Web 2.0 space is seen as something that companies want to emulate.
- *Web-based economies of scale.* Companies see that they can dramatically cut the cost of capital equipment and people by using a Web-based delivery model to communities of their customers. B2C 2.0 companies are planning to support tens of millions of customers with just hundreds of employees.
- *Flexible employment models.* The use of contract and agency staff for delivery allows flexibility and agility. Agency and contract staff can be thought of as another, specialized community and can be supported with Web 2.0 techniques, similar to customers.
- *Community creation as evangelism and support.* Customers are a business's best sales, marketing, support, and development organization. The creation of communities effectively outsources these cost centers, at zero cost. Indeed, with the inclusion of targeted advertising to the community, many of the present cost centers may become profit centers.
- *Community leader advantage.* Community dynamics are such that the first successful community is by far the most powerful and the organization that owns this community is the one which controls the space. If an organization's competitors are first in the community space they will have very significant competitive advantage.

There are five areas where Web 2.0 techniques can be used in working with customer communities to provide Business to Community:

Innovation and new product development

A large percentage of new product ideas and innovations in organizations come from suppliers and customers rather than from in-house research and development organizations. These customer-generated new product ideas are more likely to be successful as they have come from the end users of the product. Clearly, organizations that can build a system which harvests these ideas can gain significant benefits. The use of Web 2.0-based customer and supplier communities as discussion forums and marketing focus group for new product ideas and incubation is a powerful technique for gathering ideas simply and cost-effectively. Many organizations are actively investigating the use of community forums and discussion groups in the product development process.

An additional benefit of this community-based new product development is that the customers have a better understanding of the product or service delivered when they are involved in its gestation and so the customer uptake of the product will be significantly enhanced.

Marketing

Probably the best known application of Web 2.0 techniques in organizations is in the marketing departments as viral marketing. The examples of community and rich content such as video being used to generate and spread buzz about products and services are legion. There are two elements to this viral marketing, initial interest generation and then viral dissemination. The initial interest generation is best done with the use of innovative image, video and movie content; it is not unusual to get millions of downloads of a creative video within hours or days of release, and ongoing interest in a product can be sustained by including an informational or tutorial element to the content. The dissemination of this material is done by the Internet community at large using IM, e-mail, or community forums. Again this dissemination can be very rapid and widespread. Engaging video of a new product or service may be passed to millions of people in hours and surface in mainstream media such as TV or newspapers in days.

There are, however, some caveats about viral marketing: First, the target audience needs to be well understood, and even then, the material may not generate community interest and buzz. Second, an organization cannot control the spread or use of materials; the use of viral marketing techniques by the community in unanticipated ways is well-documented and can create significant issues for an organization.

Sales

The cost of sale is normally a significant part of the overall cost of a product or service. In a Business to Community 2.0 organization, the community acts as the salespeople so the cost of sale is dramatically reduced, in many cases to zero. The customers themselves act as spokespeople and salespeople for the organization. There is no need for a high pressure and high cost sales organization in community-based businesses; in many cases, it is counterproductive and will actually stall sales.

Support

Support is the second best known area for the use of Web 2.0 techniques. First, there is the use of IM and chat techniques for real-time support of products by organizations. Second, the use of image capture and video for problem communication and resolution. The use of community-based product experts and self-help discussion groups is the third and most important area: Self-help has been shown to work very well in communities and is a very low cost way of providing very high quality support. As with most society-based systems however the actual mechanics of these self-help groups is not simple and requires significant thought and expertise.

Training and education

Probably the least explored usage of Web 2.0 in the enterprise is for training and education. The availability of high quality image and video provides a very low cost and simple way of providing training and demonstration materials. This rich content, when integrated with subject matter experts in the community and active discussion groups provides a powerful and simple training and education environment. While there has been relatively little activity in the training and education area today, it is certain to grow dramatically in the future.

Organizations are learning to see their customers and employees as communities with which they have both an online and offline relationship in a multifaceted community-based sales cycle: A retail specialty store may have an in-store lecture by a national expert which creates a community online around a topic of interest moderated by that expert. As part of the online discussion, community members visit the store to see particular items of interest and then order those items online within a community rewards scheme.

Conclusion

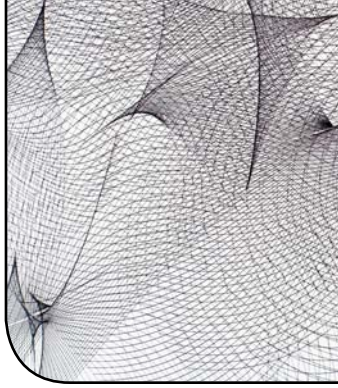
In summary, the Web 2.0 areas of rich content and community are being used successfully by organizations today, both internally for knowledge capture and reuse, and externally to create communities of customers. While most of the interest today is in the knowledge capture and reuse, there are still significant cultural and social issues to the successful implementation of these systems which are not solved by Web 2.0 techniques. The less well-explored area of the use of customer communities has much greater promise to the organization, yet comes with its own concomitant risks around IP and vandalism which have to be addressed.

Overall, the use of Web 2.0 techniques in the enterprise promises to have profound and far reaching effect on how organizations work both internally and externally, creating completely new and powerful ways of reaching, selling and supporting customers as communities.

About the Author

Michael Platt is director of Web Architecture Strategy in the architecture strategy group at Microsoft Corp. in Redmond. He has extensive experience in architecting Web and enterprise applications for large multinational enterprises, and some 30 years of experience in the IT industry. Michael has been with Microsoft for 10 years in a number of management, strategy, presales and consultancy roles. He has an Honours Degree in Electronic and Electrical Engineering and a number of published dissertations and patents.

Efficient Software Delivery through Service Delivery Platforms



by Gianpaolo Carraro, Fred Chong, and Eugenio Pace

Summary

Delivering software as a service (SaaS) has gained a lot of momentum. One reason this one-to-many delivery model is attractive is that it enables new economies of scale. Yet economy of scale does not come automatically—it has to be explicitly architected into the solution. A typical architectural pattern allowing economy of scale is "single instance multi-tenancy," and many Independent Software Vendors (ISVs) offering SaaS have moved to this architecture, with various levels of success.

There is, however, another means of improving efficiency which ISVs have not adopted with the same enthusiasm: the use of an underlying Service Delivery Platform (SDP). Adoption has been slow mainly because service delivery platforms optimized for line-of-business applications delivery are still in their infancy. But both existing and new actors in the hosting space are quickly building compelling capabilities. This paper explores the goals, capabilities, and motivations for adoption of SDP, and describes the technology and processes related to efficient software delivery through SDP.

Economies of Scale and Application Architecture

Operating systems evolved as a way of simplifying application development, operations and management. Rather than requiring each application to (re)create the full stack of subsystems needed for it to run with expected quality levels, an operating system provides an infrastructure where common, general purpose applications services are encoded and reused.

Often these services are complex technically and require expertise and specialized skills to develop. ISVs quickly understood the value proposition of an operating system and standardized their development on top of one or two operating systems. Leveraging the common underlying infrastructure allowed them to spend more time on the domain-specific part of the application, which is, in the end, what software buyers are paying for.

There are cases where the specific needs of an application go beyond what a general purpose operating system offers. Under this circumstance parts of the operating system are replaced by custom developed modules providing the specific level of support. For example, databases often write their own file systems, and in the case of Microsoft SQL Server, its own scheduler and memory management. But these very specific needs are very much the exceptions than the norm.

Enterprises and ISVs continually factor out common components from their solutions into "horizontal" application frameworks. These frameworks provide further abstracted, common, shared components (and procedures) that lead to increased reuse, increased productivity, and in general, a more predictable development and operational environment.

"THE DOWNWARD FLOW OF HORIZONTAL CAPABILITIES FROM ISV "PLUMBING" INTO FRAMEWORKS, AND THEN INTO CORE PLATFORMS CAN BE GENERALIZED FURTHER TO COVER THE SCENARIOS OF SOFTWARE DELIVERED AS A SERVICE. IN THIS SENSE, AN SDP BECOMES AN "OPERATING SYSTEM" FOR SERVICE DELIVERY."

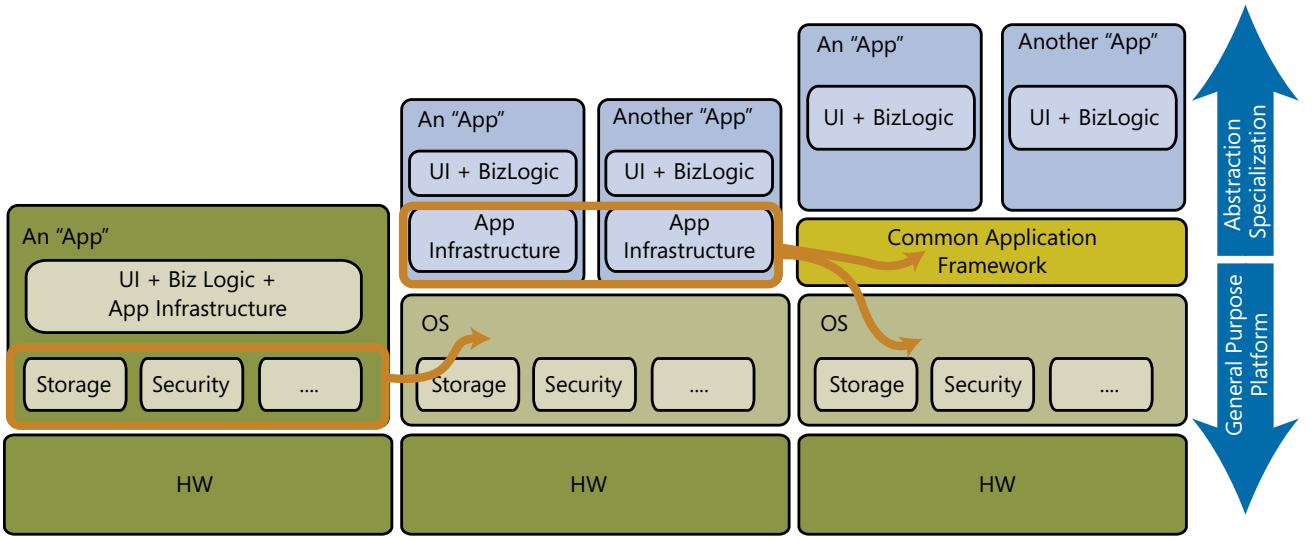
The problem with building rich frameworks is that they often do not directly contribute to the value of the software itself. They are "necessary evils" that facilitate the longer term maintenance of the application, and ISVs and enterprises would rather have someone else own them. This reinforces a win-win situation where "plumbers" can concentrate and sell horizontal functionalities so "domain experts" can concentrate on "vertical" functionalities where the value proposition for the buyer is higher. Let's face it, nobody buys a solution because it has a better exception handling mechanism.

As illustrated in Figure 1, there is a natural and ongoing process of extraction and generalization of functionality from applications into frameworks and from frameworks into core platform components. By increasing the amount of shared elements, this flow improves economies of scale.

The downward flow of horizontal capabilities from ISV "plumbing" into frameworks, and then into core platforms can be generalized further to cover the scenarios of software delivered as a service. In this sense, an SDP becomes an "operating system" for service delivery, specialized in the horizontal characteristics and requirements of SaaS-delivered applications.

Traditional hosting companies are natural candidates to implement and offer an SDP given their background, expertise, skill set, and installed base. However, other actors might as well consider entering this emerging space. ISVs that are currently self-hosting their SaaS solution could consider monetizing the self-hosting environment they

Figure 1: Factoring out commonality into a reusable platform



have built for themselves by offering it as a general-purpose SaaS delivery environment. System integrators with world-class operational excellence gained through their business-process-outsourcing offerings could leverage that knowledge in the fast-growing SaaS space.

"THE KEY POINT IS THAT AN SDP'S EFFECTIVENESS IS HIGHLY DEPENDENT ON THE ARCHETYPE SERVED. THE MORE KNOWLEDGE OF THE APPLICATION AN SDP HAS, THE GREATER ITS ABILITY TO INCREASE THE EFFICIENCY OF RUNNING AND OPERATING IT, AND THE GREATER THE DEGREE OF SHARING."

Success Factors of a SDP

The degree of success of an SDP is determined by the following attributes:

1. *Core operational capabilities:* the ability to provide strong service level agreements (SLAs) on nonfunctional requirements, such as availability (fault tolerance and uptime), scalability and disaster recovery; and on generic service features, such as multiple geography presence, 24-7 customer support, and multilayered physical security.
2. *Services depth:* the degree of sophistication of the services it provides, such as billing support for multiple payment options.
3. *Services breadth:* the completeness of the platform; in other words, the support for the different stages of a SaaS-delivered application life cycle, such as billing service, application staging, capacity planning services, or methods for patching and upgrading applications.
4. *Ease of use:* usability from the ISV perspective; in other words, the cost of learning and using the services provided by the SDP. Easy-to-use SDPs have short ramp time, complete documentation, well-planned interfaces and SDKs, sample code, templates, wizards and training content.

Note that the four attributes above are not meant to represent a maturity model. Observation of existing SDP offerings seems to indicate that two strategies are currently offered: a breadth strategy, optimized for providing a comprehensive, end-to-end platform that covers every step of a typical SaaS-delivered application life cycle (see Figure 2), with albeit shallow coverage in some capabilities; a depth strategy, focusing on certain stages only, but offering sophisticated features on these. The same survey also indicates that the "ease of use" attribute is not yet fully incorporated in the offerings as many of them rely on human intensive ad hoc processes.

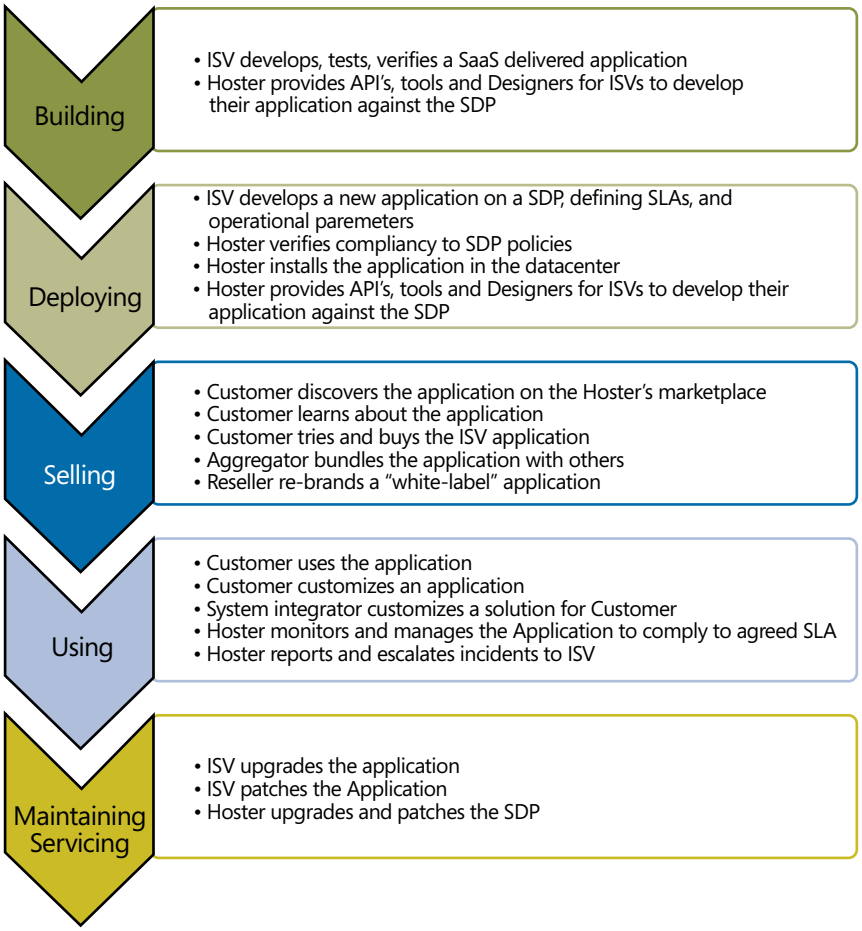
Application Archetypes: Does One Size SDP Fit All?

Business applications can be classified in different archetypes based on their characteristics and requirements. A few examples of these archetypes are:

- *Online transaction processing systems (OLTP):* characterized by low latency, high responsiveness, data integrity, predefined UI workflows. Instances of this archetype are e-commerce sites, e-banking systems, CRM systems.
- *Analysis systems or online analytic processing (OLAP):* characterized by their ability to produce complex analytical and highly customizable queries on large multidimensional datasets, with low latency responses. BI systems fall into this category.
- *Batch systems:* capable of performing operations on large datasets efficiently, coordinating jobs to maximize CPU utilization with recovery policies when exceptions occur.

Each of these application families has its own constraints, characteristics, and optimal design patterns that can be applied to solve the specific challenges they present. Very often, these challenges have conflicting

Figure 2: A typical SaaS-delivered application life cycle and associated activities



goals. For example: OLTP will optimize for low latency, whereas latency for batch systems is not as important. OLTP scales better horizontally and benefits from a stateless architecture, while batch systems scale vertically and tend to be stateful. The infrastructure and services to support each is consequently significantly different.

The key point is that an SDP's effectiveness is highly dependent on the archetype served. The more knowledge of the application an SDP has, the greater its ability to increase the efficiency of running and operating it, and the greater the degree of sharing.

The Capabilities of a Service Delivery Platform

Figure 2 illustrates a typical SaaS-delivered application life cycle. Each stage of the life cycle is characterized by a number of activities performed by the ISV, the hoster, and other actors that will become more relevant as the degree of sophistication of the SDPs increases (system integrators, value-added resellers, for example)

The four scenarios below describe the capabilities and the experiences enabled by increasingly sophisticated SDPs.

Scenario A: The Basic Service Delivery Platform

The capabilities of the simplest SDP are essentially those offered by most of traditional hosters today. These services are focused mainly on core, very generic infrastructure components such as: CPU, network access, Internet security features, and storage (disk and databases); commonly referred to as "ping, power, and pipe".

Economy of scale in this scenario is limited to the sharing of the lowest levels of infrastructure: the building for the data center and the IT infrastructure, such as servers and network equipment.

In this basic scenario, deployed ISVs applications use their own standards for application architecture and little of it is exposed to the hoster or known by it. ISVs are expected to develop and provide their own multitenant management infrastructure, security infrastructure, tenant management, and so on.

Infrastructure services requests such as registering a new domain, a new virtual directory or a Web site, creating a new database, allocating storage are often performed manually or are automated with custom scripting.

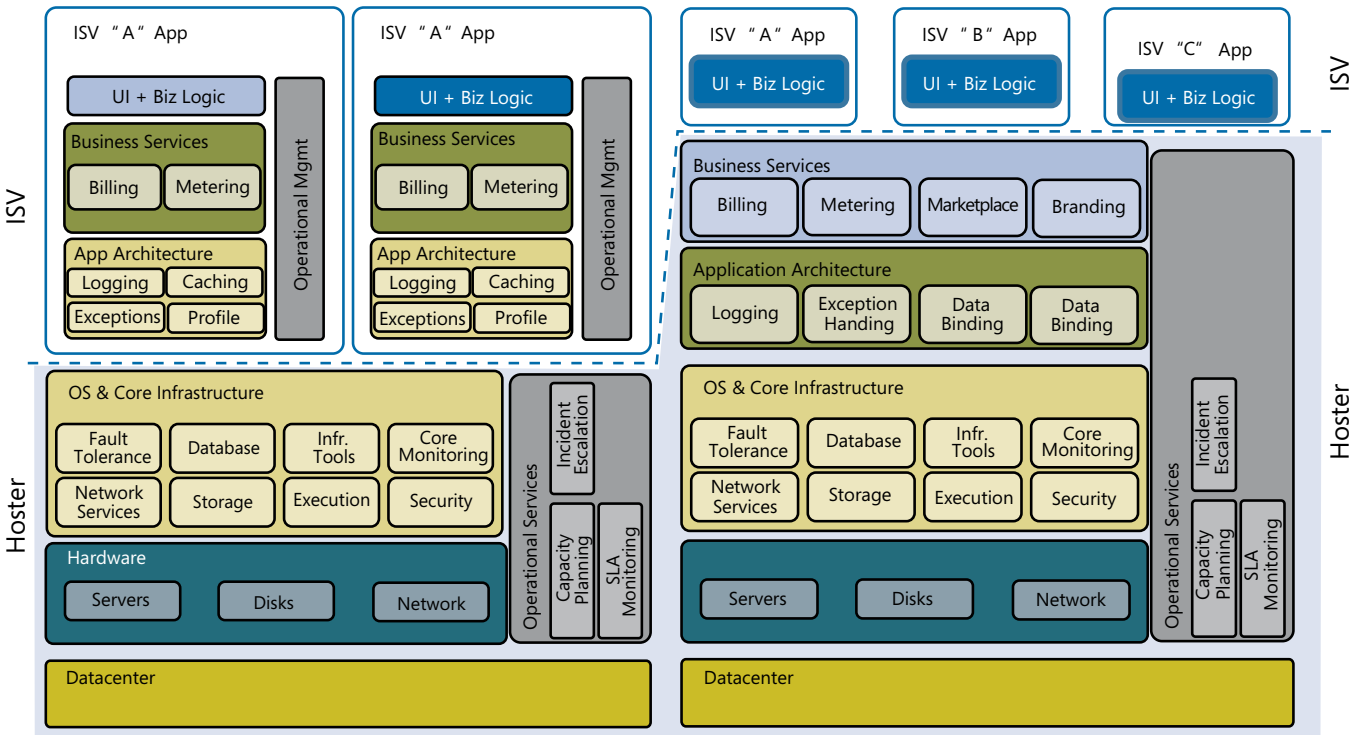
Installation and deployment of the solution require the ISV to provide the hoster with detailed instructions for configuration and setup; multiple nonstandard tuning is required (such as handling XML config files, registry keys, file paths, connection strings) because little of the application architecture is known or exposed to the hoster. These ad hoc procedures hinder the hoster from scaling its procedures to thousands of applications because it has to treat each one as an exception.

Each application is a "black box" of potentially conflicting components, competing inadequately for shared (and limited) resources, a reason many ISVs request dedicated servers that will guarantee complete isolation from other applications. However, dedicated servers goes directly against the goal of efficiency through shared infrastructure.

The hoster can only watch and act upon broad indicators to monitor the solution. They can only measure general, broad performance counters like CPU and memory workloads, bandwidth, SQL contention; and maybe some application-specific performance counters that the ISV provides on an "as needed" basis. There are little or no agreed-upon, standard mechanisms for finer-grained management of the application.

Quite remarkably though, even with very little or no knowledge of what the application does, hosters are usually able to give the ISV detailed information on database performance. This is because database artifacts are common to all ISVs (every application has tables, stored procedures, indexes, triggers, views, and so forth) and the platform where these artifacts are instantiated (the database server itself, like SQL Server) is instrumented at that level of abstraction. Hosters can therefore provide detailed contention, locking, and performance reports, and even suggest improvements on such artifacts.

Figure 3: Driving economies of scale through an increased shared infrastructure



"EVEN WITH VERY LITTLE OR NO KNOWLEDGE OF WHAT THE APPLICATION DOES, HOSTERS ARE USUALLY ABLE TO GIVE THE ISV DETAILED INFORMATION ON DATABASE PERFORMANCE BECAUSE DATABASE ARTIFACTS ARE COMMON TO ALL ISVS AND THE PLATFORM WHERE THESE ARTIFACTS ARE INSTANTIATED IS INSTRUMENTED AT THAT LEVEL OF ABSTRACTION."

The salient point here is that these artifacts exist on every database-based application regardless of the ISV authoring it. Comparatively, using a Web application as an example, the only artifact shared among different ISVs is the "Web page" identified by an URL, an element too coarse-grained to be efficiently managed. Questions, such as what part of the page takes longer to load, what components are instantiated through a particular page, or which one of these dependent components is causing contention or run-time problems, require code inspection, use of advanced tools and/or deep knowledge of how the application is built—procedures that inherently don't scale to thousands of applications.

Upgrading everything (except core operating system, networking equipment, and other core infrastructure) requires manual or ad hoc procedures as well and human interactions (e-mail, phone calls) between ISV experts and hosters.

SaaS applications operating in this basic environment are usually implemented using a wide variety of not necessarily compatible technology stacks and libraries. For example, you might find a Terminal Services hosted and delivered VB6 application, a Web app using various available frameworks and run-time components. In other words, the exception is the rule.

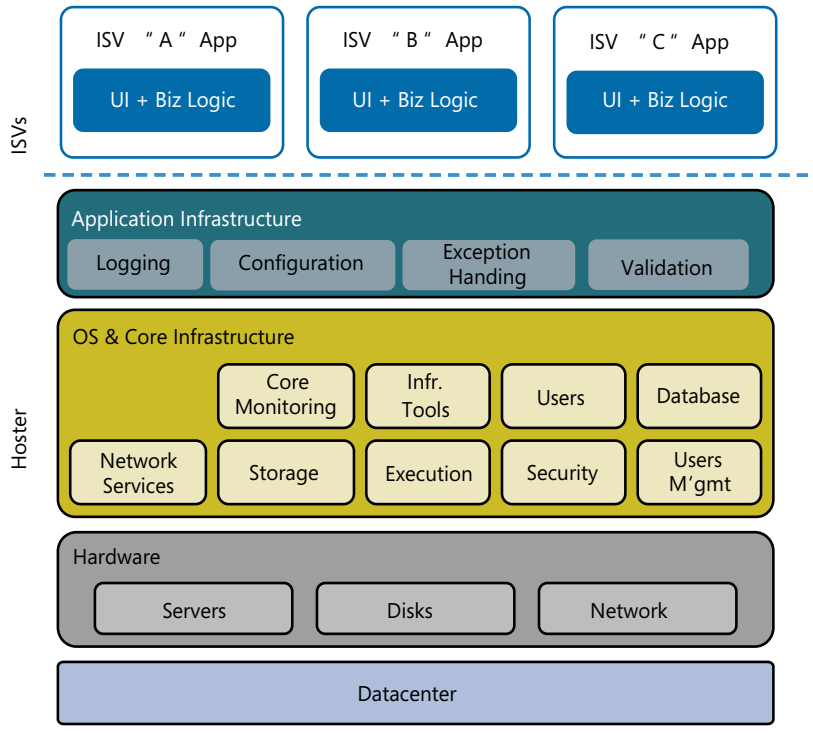
This scenario is mainly characterized by a sharing of operational infrastructure. In summary, economies of scale in this scenario are limited to fairly low-level components. One of the key factors affecting the SDP efficiency is the heterogeneity of the hosted applications.

Scenario B: Improving Efficiency Through Deeper Knowledge of the Application Architecture

An increased amount of shared components leads to higher levels of efficiency, so the question is which are the most natural candidates to be "extracted" from applications into the SDP? The obvious candidates are those referred to application infrastructure services such as: application configuration, run-time exception handling and reporting, logging and auditing, tracing, caching, data access. Every application needs them, yet they are frequently written once and again by ISVs.

An example of a common, standard and widely adopted application infrastructure framework is Microsoft's Enterprise Library.
By exposing these basic services publicly, the SDP has a much increased ability to automate common procedures and offer more advanced operational management capabilities. Thus, finer-grain tuning, customization and troubleshooting is available. Notice that the hoster

Figure 4: Increased reuse through an application infrastructure



does not need to understand in detail what the application does, but rather how it does it (for example, where are connection strings to the database stored? How are run-time exceptions logged and notified?).

Because ISVs would be developing against these APIs, the hoster is required to publish an "SDP SDK" including documentation, samples, and even some basic tools for ISVs to download and use.

Hosters in this scenario can dramatically scale out basic operational procedures as all of them are common, and exceptional cases become, well, exceptions. Applications that don't comply to the standards can of course lead to premium offerings for increased revenue streams.

Additionally, hosters can offer a higher range of differentiated services with different monetization schemes. For example, the hoster knows that all applications will log run-time exceptions using the same policies and procedures, so basic run-time exception logging and reporting could be offered in the basic hosting package, and advanced run-time exception logging, notification and escalation could become a premium offering. (See Figure 5.) Notice that with this approach the ISV application does not change, because all this logic ("plumbing") resides on the SDP side.

Scenario C: The SDP Beyond Operational Management

Scenario B is clearly an improvement over A, but the SDP can be further enhanced with the addition of more sophisticated application services, like advanced infrastructure provisioning, security-related services (user profiles, user roles, personalization) and new business services such as: business events, application metering and billing and usage intelligence.

Because contracts between services and applications can be very well defined, SLAs at the services level can now be established. Traditional SLAs at the macro level (like "uptime," "bandwidth," and "CPU utilization") are necessary but not sufficient. Much finer-grained SLAs can be negotiated and traded. For example, an SLA stating "provisioning a new tenant in less than X min" can be defined, monitored, and enforced.

The SDP SDK can be further enhanced to cover the new services provided, with stand-alone implementations of the services that enable offline development. For example, it is highly unlikely that the hoster will share with the ISV the full billing system just for development purposes. The ISV will most likely code against a "mock billing service" with equivalent interfaces and contracts and a minimal simulated implementation. The goal here is the creation of a development environment that mimics all SDP capabilities with minimal dependencies to any concrete implementation. This also avoids unnecessary exposure of SDP internals and intellectual property to the public. To support this scenario, a much deeper integration and interaction between the hoster and the ISV's own software

Figure 5: An example of differentiated offerings for the same function

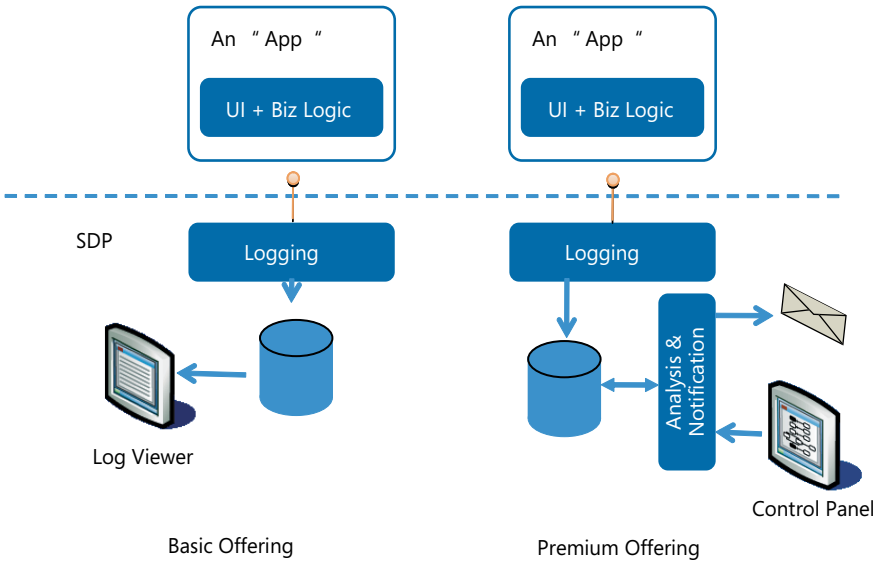
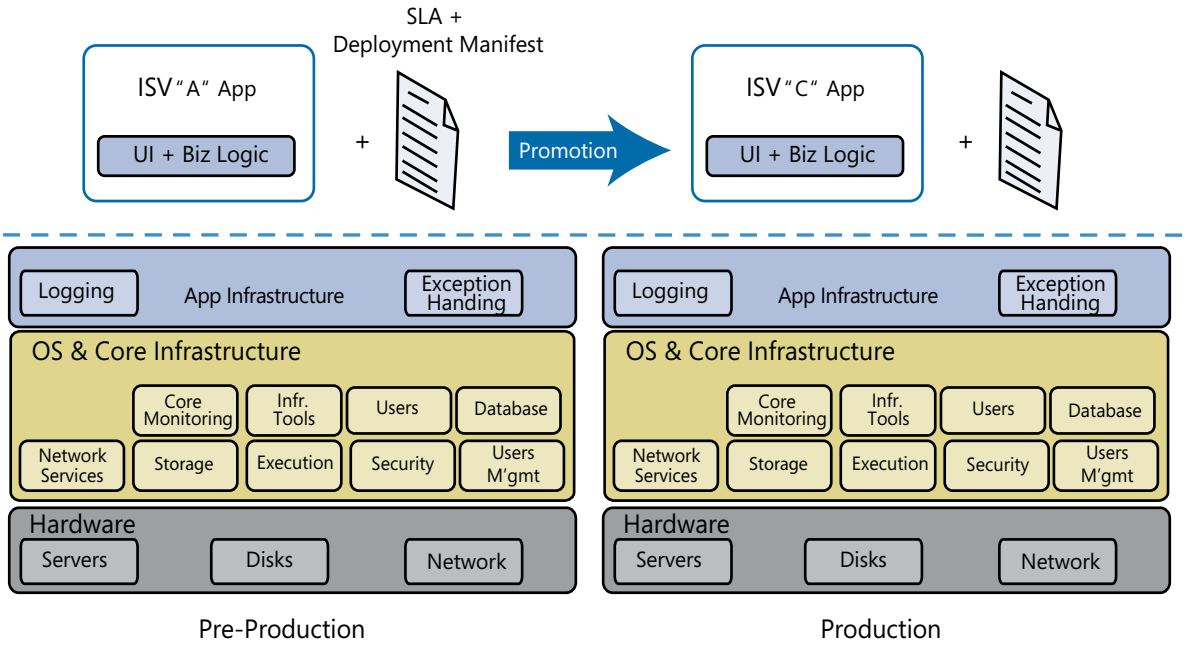


Figure 6: Deployment includes code and runtime definitions



"VERSION CONTROL, INTEGRATED INCIDENT MANAGEMENT, STAGING AND VERSION PROMOTION BETWEEN STAGING ENVIRONMENTS ARE OFFERED THROUGH THE SDP— THIS INFRASTRUCTURE ENABLES PROGRAMS LIKE "BETA TESTERS" TO RUN ON CONTROLLED REGIONS OF THE SDP, COLLECTING FEEDBACK, USAGE PATTERNS AND BUG RESOLUTION, BEFORE FEATURES ARE PUSHED INTO THE FINAL RELEASE."

development life cycle (SDLC) is required.

Also in this scenario, the deployment of solutions into the SDP is done through highly automated procedures and minimal manual intervention. These procedures include advanced features for automatic validation and configuration of the application requirements (prerequisites, server locations, connection strings, for example).

The deployment into the SDP goes beyond the code and configuration scripts to include the negotiated SLAs, operational requirements such as automatic capacity management and incident escalation, billing parameterization.

The SDP is also able to provide staging environments, where the application can run and be verified prior to being promoted to production. Staging environments allow simulation of billing, tenant creation, faults, and so on, to enable more complex modeling of real world scenarios. These can be seen as "unit tests" for the SLAs between the hoster and ISV. (See Figure 6.)

Interestingly this leads to a new breed of services to be offered beyond the runtime: for example, fault simulation, load testing, performance analysis, optimization could be available for the ISVs. Smaller ISVs could have access to temporary resources that would be too costly for them to own.

In production, because of the deeper knowledge of how the application works, the SDP can offer automatic resource management capabilities and tuning. More advanced SDPs can dynamically assign new machines, add CPUs to a cluster, assign higher communication bandwidth, and whatever other infrastructure is required to keep the application at the agreed-upon SLAs.

As applications are fully instrumented, hosters can offer intelligence on usage patterns and finer-grained analysis of how the application performs and works, feeding back this information (possibly as a premium service) to the ISVs' product planning pipeline.

Because of the highly automated quality assurance and deployment procedures, ISVs have an opportunity to offer almost real-time product enhancements based on the intelligence collected and improve their products constantly based on more accurate user feedback.

Selling, branding, bundling, and aggregation of applications are enabled through well-defined composition rules, therefore, the hoster can create bundled offerings with common services across different solutions (such as Single Sign On or common role management). Also, "white label" branding is enabled for aggregators and third parties to create specialized, customized offerings based on the same code base.

Scenario D: The Ultimate SDP

The ultimate SDP doesn't exist yet. This section is a little bit of extrapolation and a little bit of speculation. In the most advanced scenario, in addition to all the features described above, the SDP includes services for complete application life cycle management,

allowing further integration of development, versioning, deployment, management, operations and support of an SaaS-delivered solution, allowing an extended ecosystem to contribute and collaborate to deliver specialized solutions.

Discovering, learning, trying, developing, extending, versioning, tuning, are use cases supported by the SDP. This is a full integration of the ISV Software Development Life Cycle (SDLC) and the SDP Software Operations Life Cycle (SOLC). In this scenario, aspects of software development are offered as a service itself: bug tracking, software version control, performance testing, and so on. (Figure 7.)

All SDP capabilities are expressed in machine-readable formats. Models of software components and SDP capabilities are fully integrated into the tools. Verification, analysis and simulation of these models can be performed by independent members of ecosystem: ISVs, third parties, aggregators, enterprises, and others collaborate to create complex systems.

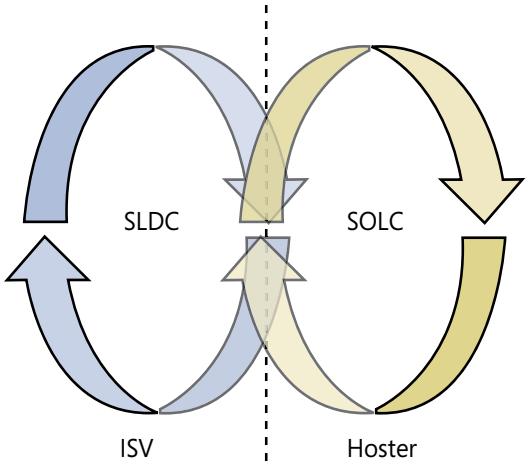
Application metadata includes not just operational parameters, but also information required to publicize the application in the SDP Marketplace (the equivalent of the "Add New Software" application registry found on Windows), provide "learn more" content, guidance content, documentation, training, additional value added professional services, and so on.

Version control, integrated incident management, staging and version promotion between staging environments are offered through the SDP, allowing ISVs, aggregators, system integrators to develop, customize and maintain different concurrent offerings. This infrastructure enables programs like "beta testers" and "early adopters" to run on controlled regions of the SDP, collecting feedback, usage patterns and bug resolution, before features are pushed into the final release.

The Marketplace service provided by the SDP offers a mechanism to discover, learn and try new offerings and is made available utilizing the metadata that accompanies every application.

As said, it will take some time for these advanced SDPs to become mainstream.

Figure 7: Integration of ISV's SDLC and hoster operational life cycle in the "Ultimate SDP"



Conclusion

SDPs represent an exciting new opportunity for traditional hosters to create differentiated, high value offerings; lowering the bar of entry for a larger number of ISVs to offer solutions with world-class operational levels. First movers into this new market category will attract large numbers of ISVs, especially those in niche or small- to medium-sized business segments that cannot economically self-host due to their skill set or the financial feasibility of their business models.

Microsoft's Architecture Strategy Team is actively investing in developing architecture guidance to help ISVs, hosters and enterprises realize the benefits of software and services, leveraging the Microsoft Platform.

Resources

MSDN Architecture Development Center
http://msdn.microsoft.com/architecture

SaaS Section on MSDN Dev Center
http://msdn.microsoft.com/architecture/saas

LitwareHR – A sample SaaS-delivered application developed by Microsoft Architecture Strategy Team
http://www.codeplex.com/litwarehr

About the Authors

Gianpaolo Carraro, director of Service Delivery – Microsoft Architecture Strategy, drives SaaS thought leadership and architectural best practices for Microsoft. Prior to Microsoft, Gianpaolo was cofounder and chief architect of a SaaS startup; he is a former member of technical staff at Bell Laboratories. Gianpaolo is a published author and a frequent speaker at major international IT conferences. You can learn more about him through his blog: http://blogs.msdn.com/gianpaolo

Fred Chong, architect - Microsoft Architecture Strategy Team, is recognized by the industry as a subject matter expert on the topic of SaaS architecture. Previously, he has designed and implemented security protocols and networking components for Microsoft products and customer solutions. Fred has also conducted research at the IBM T.J. Watson Research Center and the University of California at San Diego. You can find his blog at http://blogs.msdn.com/fred_chong

Eugenio Pace, architect - Microsoft Architecture Strategy Team, is responsible for developing architecture guidance in the SaaS space. Before joining AST, he was a product manager in the patterns and practices team at Microsoft where he was responsible for delivering client-side architecture guidance, including Web clients, smart clients, and mobile clients. During that time, his team shipped the Composite UI Application Block, and three software factories for mobile and desktop smart clients and for Web development. Before joining patterns and practices, he was an architect at Microsoft Consulting Services where he led the development of several enterprise solutions throughout the world. You can find his blog at http://blogs.msdn.com/eugeniop

Secure Cross-Domain Communication in the Browser

by Danny Thorpe

Summary

A shopper can walk into virtually any store and make a purchase with nothing more than a plastic card and photo ID. The shopper and the shopkeeper need not share the same currency, nationality, or language. What they do share is a global communications system and global banking network that allows the shopper to bring their bank services with them wherever they go and provides infrastructure support to the shopkeeper. What if the Internet could provide similar protections and services for Web surfers and site keepers to share information?

Developing applications that live inside the Web browser is a lot like window shopping on Main Street: Lots of stores to choose from, lots of wonderful things to look at in the windows of each store, but you can't get to any of it. Your cruel stepmother, Frau Browser, yanks your leash every time you lean too close to the glass. She says it's for your own good, but you're beginning to wonder if your short leash is more for her convenience than your safety.

Web browsers isolate pages living in different domains to prevent them from peeking at each other's notes about the end user. In the early days of the Internet, this isolation model was fine because few sites placed significant application logic in the browser client, and even those that did were only accessing data from their own server. Each Web server was its own silo, containing only HTML links to content outside itself.

That's not the Internet today. The Internet experience has evolved into aggregating data from multiple domains. This aggregation is driven by user customization of sites as well as sites that add value by bringing together combinations of diverse data sources. In this world, the Web browser's domain isolation model becomes an enormous obstacle hindering client-side Web application development. To avoid this obstacle, Web app designers have been moving more and more application logic to their Web servers, sacrificing server scalability just to get things done. Meanwhile, the end user's 2GHz, 2GB dumb terminal sits idle.

If personal computers were built like a Web browser, you could save your data to disk, but you couldn't use those files with any other application on your machine, or anyone else's machine. If you decided to switch to a different brand of photo editor, you wouldn't be able to edit any of your old photos. If you complained to the makers of your old photo editor, they would sniff and declare "We don't know what that other photo editor might do with your data. Since we don't know or trust that other photo editor, then neither should you! And no, we won't let you use 'your' photos with them, because since we're providing the storage space for those photos, they're really partly our photos."

You couldn't even find your files unless you knew first which application you created them with. "Which photo editor did I use for Stevie's birthday photos? I can't find them!"

And what happens when that tragically hip avante-garde photo editor goes belly up, never to be seen again? It takes all your photos with it!

Sound familiar? It happens to all of us every day using Internet Web sites and Web applications. Domain isolation prevents you from using your music playlists to shop for similar tunes at an independent online store (unrelated to your music player manufacturer) or at a kiosk within a retail store.

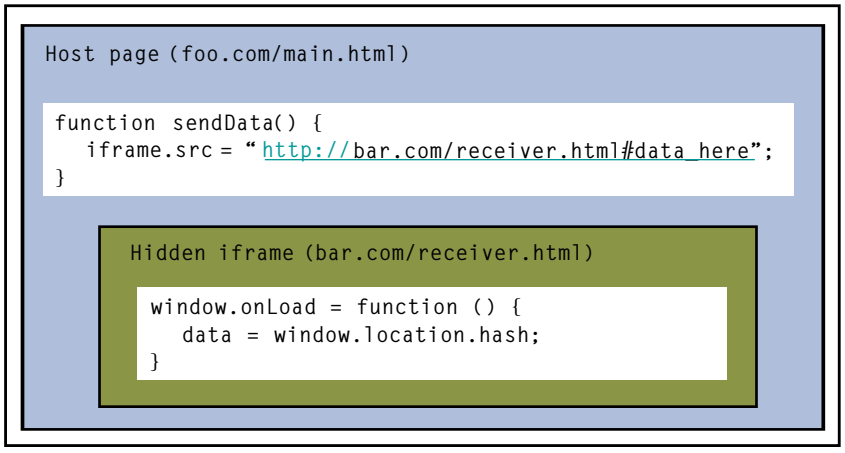
Domain isolation also makes it very difficult to build lightweight low-infrastructure Web applications that slice and dice data drawn from diverse data servers within a corporate network. A foo.bar.com subdomain on your internal bar.com corpnet is just as isolated from bar.com and bee.bar.com as it is from external addresses like xyz.com.

Nevertheless, you don't want to just tear down all the walls and pass around posies. The threats to data and personal security that the browser's strict domain isolation policy protects against are real, and nasty. With careful consideration and infrastructure, there can be a happy medium that provides greater benefit to the user while still maintaining the necessary security practices. Users should be in control of when, what, and how much of their information is available to a given Web site. The objective here is not free flow of information in all directions, but freedom for users to use their data where and when it serves their purposes, regardless of where their data resides.

What is needed is a way for the browser to support legitimate cross-domain data access without compromising end user safety and control of their data.

One major step in that direction is the developing standards proposal organized by Ian Hickson to extend xmlHttpRequest to support cross-domain connections using domain-based opt-in/opt-out by the server being requested. (See Resources.) If this survives peer review and if it is implemented by the major browsers, it offers hope of diminishing the cross-domain barrier for legitimate uses,

Figure 1: iframe URL data passing



while still protecting against illegitimate uses. Realistically, though, it will be years before this proposal is implemented by the major browsers and ubiquitous in the field.

What can be done now? There are patterns of behavior supported by all the browsers which allow JavaScript code living in one browser domain context to observe changes made by JavaScript living in another domain context within the same browser instance. For example, changes made to the width or height property of an iframe are observable inside as well as outside the iframe. Another example is the iframe.src property. Code outside an iframe cannot read the iframe's src URL property, but it can write to the iframe's src URL. Thus, code outside the iframe can send data into the iframe via the iframe's URL.

This URL technique has been used by Web designers since iframes were first introduced into HTML, but uses are typically primitive, purpose-built, and hastily thrown together. What's worse, passing data through the iframe src URL can create an exploit vector, allowing malicious code to corrupt your Web application state by throwing garbage at your iframe. Any code in any context in the browser can write to the iframe's .src property, and the receiving iframe has no idea where the URL data came from. In most situations, data of unknown origin should never be trusted.

This article will explore the issues and solution techniques of the secure client-side cross-domain data channel developed by the Windows Live Developer Platform group.

IFrame URL technique

An iframe is an HTML element that encapsulates and displays an entire HTML document inside itself, allowing you to display one HTML document inside another. We'll call the iframe's parent the outer page or host page, and the iframe's content the inner page. The iframe's inside page is specified by assigning a URL to the iframe's src property.

When the iframe's source URL has the same domain name as the outer, host page, JavaScript in the host page can navigate through the iframe's interior DOM and see all of its contents. Conversely, the iframe can navigate up through its parent chain and see all of its DOM siblings in the host page and their properties. However, when the iframe's source URL has a domain different from the host page, the host cannot see the iframe's contents, and the iframe cannot see the host page's contents.

Even though the host cannot read the iframe element's src property, it can still write to it. The host page doesn't know what the iframe is currently displaying, but it can force the iframe to display something else.

Each time a new URL is assigned to the iframe's src property, the iframe will go through all the normal steps of loading a page, including firing the onLoad event.

We now have all the pieces required to pass data from the host to the iframe on the URL. (See Figure 1.) The host page in domain foo.com can place a URL-encoded data packet on the end of an existing document URL in the bar.com domain. The data can be carried in the URL as a query parameter using the ? character (http://bar.com/receiver.html?datadatadata) or as a bookmark using the # character (http://bar.com/receiver.html#datadatadata). There's a big difference between these two URL types which we'll explore in a moment.

"DOMAIN ISOLATION MAKES IT VERY DIFFICULT TO BUILD LIGHTWEIGHT LOW-INFRASTRUCTURE WEB APPLICATIONS THAT SLICE AND DICE DATA DRAWN FROM DIVERSE DATA SERVERS WITHIN A CORPORATE NETWORK. NEVERTHELESS, YOU DON'T WANT TO JUST TEAR DOWN ALL THE WALLS AND PASS AROUND POSIES."

The host page assigns this url to the iframe's src property. The iframe loads the page and fires the page's onLoad event handler. The iframe page's onLoad event handler can look at its own URL, find the embedded data packet, and decode it to decide what to do next.

That's the iframe URL data passing technique at its simplest. The host builds a URL string from a known document url + data payload, assigns it to the src property of the iframe, the iframe "wakes up" in the onLoad event handler and receives the data payload. What more could you ask for?

A lot more, actually. There are many caveats with this simple technique:

- *No acknowledgement of receipt.* The host page has no idea if the iframe successfully received the data.
- *Message overwrites.* The host doesn't know when the iframe has finished processing the previous message, so it doesn't know when it's safe to send the next message.
- *Capacity limits.* A URL can only be so long, and the length limit varies by browser family. Firefox supports URLs as long as 40k or

so, but IE sets the limit at less than 4k. Anything longer than that will be truncated or ignored.

- *Data has unknown origin.* The iframe has no idea who put the data into its URL. The data might be from our friendly foo.com host page, or it might be evil.com lobbing spitballs at bar.com hoping something will stick or blow up.
- *No replies.* There's no way for script in the iframe to pass data back to the host page.
- *Loss of context.* Because the page is reloaded with every message, the iframe inner page cannot maintain global state across messages

"SHOULD WE USE ? OR # TO TACK DATA ONTO THE END OF THE IFRAME URL? THOUGH INNOCUOUS ENOUGH ON THE SURFACE, THERE ARE ACTUALLY A FEW SIGNIFICANT DIFFERENCES IN HOW THE BROWSERS HANDLE URLS WITH QUERY PARAMS VERSUS URLS WITH BOOKMARKS."

Hiding Data In Bookmarks

Should we use ? or # to tack data onto the end of the iframe URL? Though innocuous enough on the surface, there are actually a few significant differences in how the browsers handle URLs with query params versus URLs with bookmarks. Two URLs with the same base path but different query params are treated as different URLs. They will appear separately in the browser history list, will be separate entries in the browser page cache, and will generate separate network requests across the wire.

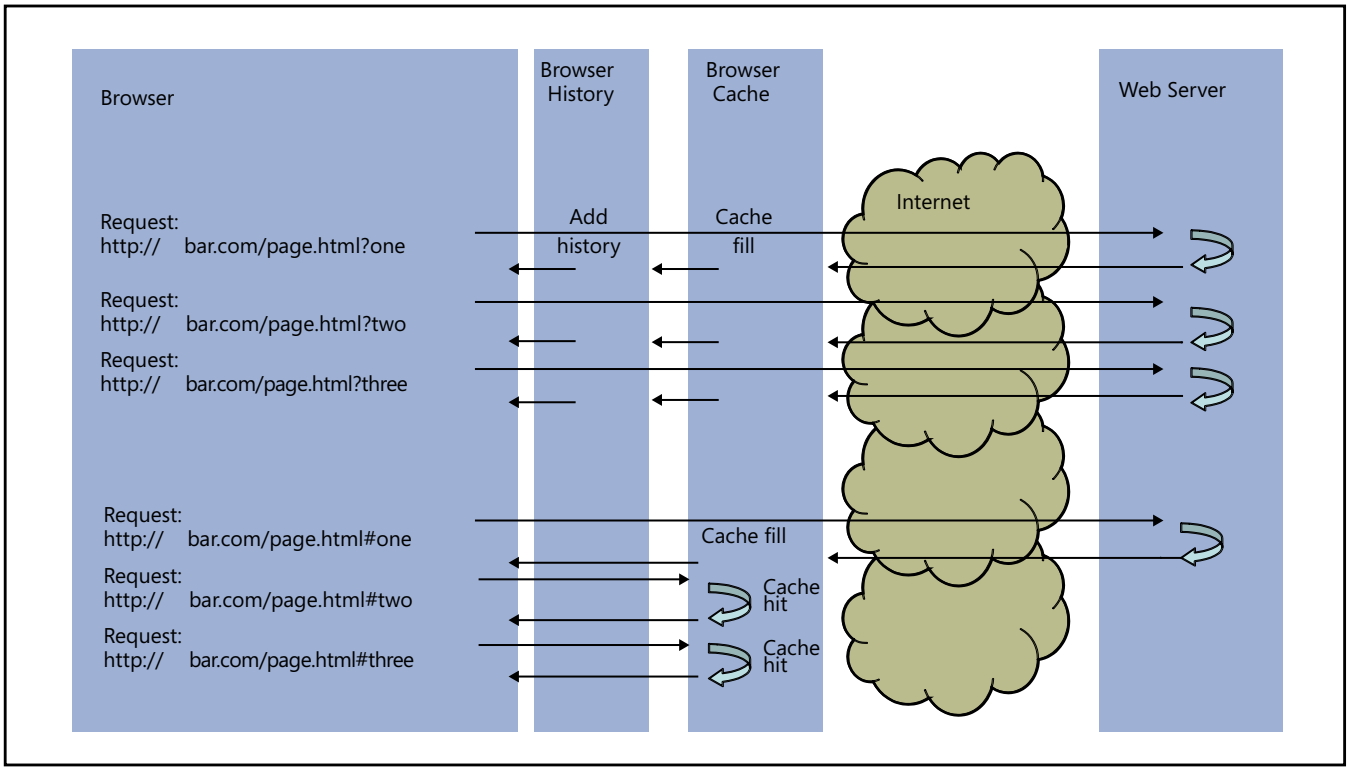
URL bookmarks were designed to refer to specially marked anchor tags within a page. The browser considers two URLs with the same base path but with different bookmark text after the # char to be the same URL as far as browser history and caches are concerned. The different bookmarks are just pointing to different parts of the same page (URL), but it's the same page nonetheless.

The URLs http://bar.com/page.html#one, http://bar.com/page.html#two, and http://bar.com/page.html#three are considered by the browser to be cache-equivalent to http://bar.com/page.html. If we used query params, the browser would see three different URLs and three different trips across the network wire. Using bookmarks, though, we have at most one trip across the network wire—subsequent requests will be filled from the local browser cache. (See Figure 2.)

For cases where we need to send a lot of messages across the iframe URL using the same base URL, bookmarks are perfect. The data payloads in the bookmark portion of the URL will not appear in the browser history or browser page cache. What's more, the data payloads will never cross the network wire after the initial page load is cached!

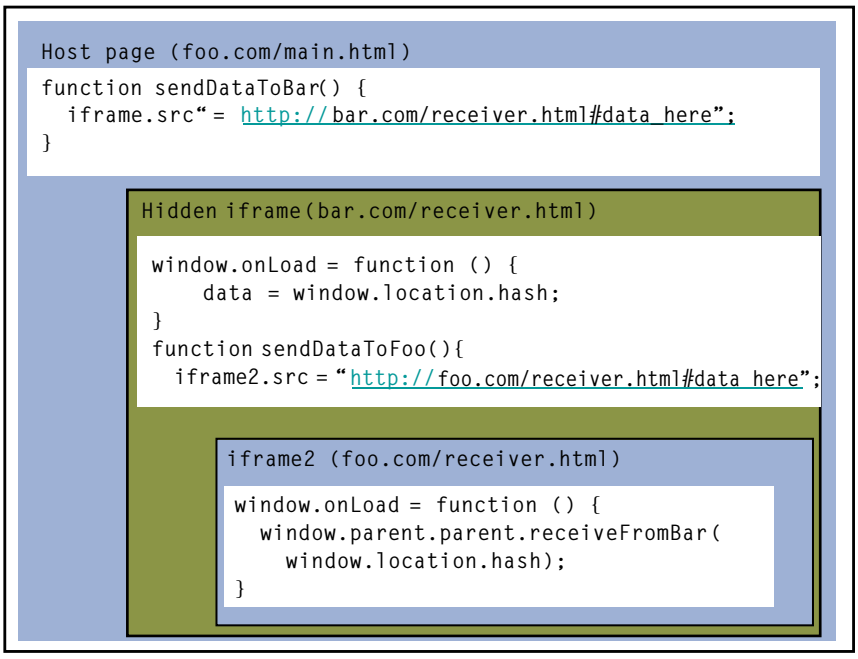
The data passed between the host page and the iframe cannot be viewed by any other DOM elements on the host page because the

Figure 2: Cache equivalence of bookmark URLs



iframe is in a different domain context from the host page. The data doesn't appear in the browser cache, and the data doesn't cross the network wire, so it's fair to say that the data packets are observable only by the receiving iframe or other pages served from the bar.com domain.

Figure 3: Message in a Klein Bottle



Sender Identification

Perhaps the biggest security problem with the simple iframe URL data passing technique is not knowing with confidence where the data came from. Embedding the name of the sender or some form of application ID is no solution, as those can be easily copied by impersonators. What is needed is a way for a message to implicitly identify the sender in such a way that could not be easily copied.

The first solution that pops to mind for most people is to use some form of encryption using keys that only the sender and receiver possess. This would certainly do the job, but it's a rather heavy handed solution, particularly when JavaScript is involved.

There is another way, which takes advantage of the critical importance of domain name identity in the browser environment. If I can send a secret message to you using your domain name, and I later receive that secret as part of a data packet, I can reasonably deduce that the data packet came from your domain.

The only way for the secret to come from a third-party domain is if your domain has been compromised, the user's browser has been compromised, or my DNS has been compromised. All bets are off if your domain or your browser have been compromised. If DNS poisoning is a real concern, you can use https to validate that the server answering requests for a given domain name is in fact the legitimate server.

If the sender gives a secret to the receiver, and the receiver gives a secret to the sender, and both secrets are carried in every data packet sent across the iframe URL data channel, then both parties can have confidence in the origin of every message. Spitballs thrown in by evil.com can be easily recognized and discarded. This exchange of secrets is inspired by the SSL/https three-phase handshake.

These secrets do not need to be complex or encrypted, since the data packets sent through the iframe URL data channel are not visible to any third party. Random numbers are sufficient as secrets, with one caveat: The JavaScript random number generator (Math.random()) is not cryptographically strong, so it is a risk for producing predictable number sequences. Firefox provides a cryptographically strong random number generator (crypto.random()), but IE does not. As a result, in our implementation we opted to generate strong random numbers on the Web server and send them down to the client as needed.

Sending to the Sender

Most of the problems associated with the iframe URL data passing technique boil down to reply generation. Acknowledging packets requires the receiver to send a reply to the sender. Exchanging secrets requires replies in both directions. Message throttling and breaking large data payloads into multiple smaller messages require receipt acknowledgement.

So how can the iframe communicate back up to the host page? Not by going up, but by going down. The iframe can't assign to anything in its parent because the iframe and the parent reside in different domain contexts. But the bar.com iframe (A) can contain another iframe (B) and A can assign to B's src property a URL in the domain of the host page (foo.com). foo.com host page contains bar.com iframe (A) contains foo.com iframe (B).

Great, but what can that inner iframe do? It can't do much with its parent, the bar.com iframe. But go one more level up and you hit paydirt: B's parent's parent is the host page in foo.com. B's page is in foo.com, B.parent.parent is in foo.com, so B can access everything in the host page and call JavaScript functions in the host page's context.

The host page can pass data to iframe A by writing a URL to A's src property. A can process the data, and send an acknowledgement to the host by writing a URL to B's src property. B wakes up in its onLoad event and passes the message up to its parent's parent, the host page. Voila. Round trip acknowledgement from a series of one-way pipes connected together in a manner that would probably amuse Felix Klein, mathematician and bottle washer.

Stateful Receiver

To maintain global state in the bar.com context across multiple messages sent to the iframe, use two iframes with bar.com pages. Use one of the iframes as a stateless message receiver, reloading and losing its state with every message received. Place the stateful application logic for the bar.com side of the house in the other iframe. Reduce the messenger iframe page logic to the bare

"IF I CAN SEND A SECRET MESSAGE TO YOU USING YOUR DOMAIN NAME, AND I LATER RECEIVE THAT SECRET AS PART OF A DATA PACKET, I CAN REASONABLY DEDUCE THAT THE DATA PACKET CAME FROM YOUR DOMAIN. IF DNS POISONING IS A CONCERN, YOU CAN USE HTTPS TO VALIDATE THAT THE SERVER ANSWERING REQUESTS FOR A GIVEN DOMAIN NAME IS IN FACT THE LEGITIMATE SERVER."

minimum required to pass the received data to the stateful bar.com iframe.

An iframe cannot enumerate the children of its parent to find other bar.com siblings, but it can look up a sibling iframe using window.parent.frames[] if it knows the name of the sibling iframe. Each time it reloads to receive new data on the URL, the messenger iframe can look up its stateful bar.com sibling iframe using window.parent.frames[] and call a function on the stateful iframe to pass the new message data into the stateful iframe. Thus, the bar.com domain context in browser memory can accumulate message chunks across multiple messages to reconstruct a data payload larger than the browser's maximum URL length.

Application of Ideas

The Windows Live Developer Platform team has developed these ideas into a JavaScript "channel" library. These cross-domain channels are used in the implementation of the Windows Live Contacts and Windows Live Spaces Web controls (http://dev.live.com), intended to reside on third party Web pages but execute in a secure iframe in the live.com domain context. The controls provide third party sites with user-controlled access to their Windows Live data such as the user's contacts list or Spaces photo albums. The channel object supports sending arbitrarily large data across iframe domain boundaries with receipt acknowledgement, message throttling, message chunking, and sender identification all taking place under the hood.

Our goal is to groom this channel code into a reusable library, available to internal Microsoft partners as well as third party Web developers. While the code is running well in its current contexts, we still have some work to do in the area of self-diagnostics and troubleshooting—when you get the channel endpoints configured correctly, it works great, but it can be a real nightmare to figure out what isn't quite right when you're trying to get it set up the first time. The main obstacle is the browser itself—trying to see what's (not) happening in different domain contexts is a bit of a challenge when the browser won't show you what's on the other side of the wall.

User Empowerment

Hardly 40 years ago, a shopper on Main Street USA had to go to considerable effort to convince a shopkeeper to accept payment. If you didn't have cash (and lots of it), you were most likely out of luck. If you had foreign currency, you'd need to find a big bank in a big city to exchange for local currency. Checks from out of town were rarely accepted, and store credit was offered only to local residents.

Today, shoppers and shopkeepers share a global communications system and global banking network that allows the shopper to bring their bank services with them wherever they go, and helps the shopkeeper make sales they otherwise might miss. The banking network also provides infrastructure support to the shopkeeper, helping with currency conversion, shielding from credit risk and reducing losses due to fraud.

Now, why can't the Internet provide similar protections and empowerments for the wandering Web surfer, and infrastructure services for Web site keepers? Bring your data and experience with you as you move from site to site (the way a charge card brings your banking services with you as you shop), releasing information to the site keepers only at your discretion. The Internet is going to get there—it's just a matter of how well and how soon.

Acknowledgments

Kudos to Scott Issacs for the original iframe URL data passing concept. Many thanks to Yaron Goland and Bill Zissimopoulos for their considerable contributions to the early implementations and debugging of the channel code, and to Gabriel Corverra and Koji Kato for their work in the more recent iterations. "*It's absolute insanity, but it just might work!*"

Resources

XMLHttpRequest 2, Ian Hickson
http://www.mail-archive.com/public-webapi@w3.org/msg00341.html
http://lists.w3.org/Archives/Public/public-webapi/2006Jun/0012.html

Anne van Kesteren's weblog
http://annevankesteren.nl/2007/02/xxx

About the Author

Danny Thorpe is a developer on the Windows Live Developer Platform team. His badge says Principal SDE, but he prefers "Windows Live Quantum Mechanic" since he spends much of his time coaxing stubborn little bits to migrate across impenetrable barriers. In past lives he worked on "undisclosed browser technology" at Google, and before that he was a Chief Scientist at Borland and Chief Architect of the Delphi compiler. At Borland he had the good fortune to work under the mentorship of Anders Hejlsberg, Chuck Jazdzewski, Eli Boling and many other Borland legends. Prior joining to Borland, he was too young to remember much. Check out his blog at http://blogs.msdn.com/dthorpe.

An Application-Oriented Model for Relational Data

by Michael Pizzo

Summary

Most applications deal with data of some type or another, where the source of the data often resides in a database. Yet for a variety of reasons the shape of the data in the database is often different from the model that the application interacts with. This article describes working with data in a database through a virtual "Conceptual Model" more appropriate for the application.

Application Models versus Storage Schemas

Modern applications, web applications in particular, are fundamentally about exposing and manipulating data of one type or another. The data may be in the form of search results, inventory catalog, user profile, account information, financial information, personnel information, map coordinates, weather, etc., but it's all data, and it's typically stored in a database somewhere.

However, the data stored in the database is generally not in the most appropriate form for the application to manipulate or expose to the user. The relational data model of the database schema is typically (and rightly) optimized around storage and integrity concerns, not for application usage.

As Dr. Peter Chen explains in his groundbreaking paper introducing the Entity-Relationship Model, "The relational model…can achieve a high degree of data independence, but it may lose some important semantic information about the real world." His paper goes on to describe an alternate Entity-Relationship Model that "…adopts the more natural view that the real world consists of entities and relationships." (See Resources)

Simply put, today's object-oriented applications, not to mention end-users, tend to reason about data in richer terms than the flat rows and columns of a relational database. The "real world" includes a strong notion of the type of the object, its identity, and its relationships with other objects.

Putting aside the expressivity issues for a moment, even if all of the application concepts could be represented through the relational model, the application writer often doesn't have control over the database schema. Even worse, the schema could change over time in order to optimize different usage patterns, invalidating hard-coded access paths, mappings, and implicit assumptions.

Small applications typically start out directly embedding the logic to map relational schema to application data objects. As the application grows, or for applications that are built from the start as part of a larger enterprise framework, the data access logic is commonly broken out into a separate Data Abstraction Layer, or DAL. Whether part of the application or a separate component, hard-coding implicit assumptions about database schema, relationships, usage conventions, and access patterns make it increasingly difficult to maintain or extended this data access code over time, especially in light of changes to the underlying schema.

Let's take a look at some of these problems in a little more detail.

Database Schema Normalization

Data in a database is generally laid out in a "normalized" view as described by Dr. Codd, with separate, homogenous (rectangular) tables containing columns of single scalar values. Redundancy is reduced in order to improve integrity of the data through moving non-row specific values into a separate table. Data from these individual tables is combined through joins, based on implicit application knowledge about what different values within the columns represent. Foreign keys may or may not be used between tables of related information as a means of further enforcing data integrity, but do not themselves define navigation paths or join conditions.

> "TODAY'S OBJECT-ORIENTED APPLICATIONS, NOT TO MENTION END-USERS, TEND TO REASON ABOUT DATA IN RICHER TERMS THAN THE FLAT ROWS AND COLUMNS OF A RELATIONAL DATABASE."

Let's take an example. Suppose you ran a marina that sold used watercraft, and you wanted to track your inventory in a database, exposed to customers through a web application. The information you have to store for each boat is: Registration, Make, Year, Length and Beam (width). For boats with engines you want to store Make, Model, Year, Horsepower, type of fuel, and SerialNumber of the engine(s). A fully normalized schema might break the engine information into three separate tables, one containing the information for a particular type of motor (Make, Model, Horsepower and fuel type, with Make and Model comprising a Composite Key), one for each actual engine (SerialNumber, Year, Make, and Model) and one that associated boats and engines. The resulting schema might look something like Figure 1.

In order to show a relatively simple inventory page containing boat registration number, year, and make, along with associated engine

Figure 1: Fully Normalized Schema

Boats				
RegNum	**Year**	**Make**	**Length**	**Beam**
WN123AB	1977	Hunter	25	8
WN234CD	1999	Calabria	23	8'7"
WN345EF	1962	Del Mar	16	6'
WN456GH	1957	Harvey	13.5	5'10"
WN567IJ	1997	Seadoo	9	3'10"
WN678KL	1996	Bayliner	47	14'11"
...	...	...	...	...

EngineTypes			
Make	**Model**	**HP**	**Fuel**
Clinton	K990	9.9	Gas
Mercruiser	350MagMPI	300	Gas
Mercury	Mark30	30	Gas
Tohatsu	M50CEPTS	50	Gas
Rotax	720CC	85	Gas
Hino	W06DTA	310	Diesel
...	...	...	...

Engines			
SerialNum	**Year**	**Make**	**Model**
C1075	1975	Clinton	K990
M30099	1999	Mercruiser	350MagMPI
M3060	1962	Mercury	Mark30
T5090	1990	Tohatsu	M50CEPTS
R8596	1997	Rotax	720CC
H31096A	1996	Hino	W06DTA
H31096B	1996	Hino	W06DTA
...	...	...	...

Boat Engines	
EngineSerialNum	**BoatID**
C1075	WN123AB
M30099	WN234CD
M3060	WN345EF
T5090	WN456GH
R8596	WN567IJ
H31096A	WN678KL
H31096B	WN678KL
...	...

information, for all motor boats, your web application may use the following query:

```
SELECT Boats.RegNum, Boats.Year AS Boat_Year,
 Boats.Make AS Boat_Make, BoatEngine.SerialNumber,
 BoatEngine.Year AS Engine_Year, BoatEngine.Make
 AS Engine_Make, BoatEngine.Model AS Engine_Model,
 BoatEngine.HP
FROM Boats
 INNER JOIN (
  SELECT EngineType.Make, BoatEngines.BoatID,
                                EngineTypes.HP
  FROM EngineTypes
   INNER JOIN (Engines
    INNER JOIN BoatEngines
    ON Engines.SerialNumber =
                     BoatEngines.EngineSerialNum)
  ON EngineTypes.Model = Engines.Model
   AND EngineTypes.Make = Engines.Make
) AS BoatEngine
ON Boats.RegNum = BoatEngine.BoatID
```

This query is not only fairly complex, but requires (and embeds) an implicit understanding of how the tables are related; that the Make and Model columns of the Engines table relate to the Make and Model columns of the EngineTypes table, and that the EngineSerialNum and BoatID columns of the BoatEngines table relate to the SerialNum and RegNum columns of the Engines and Boats tables, respectively.

Also, the query attempts to return only motorboats (and not sailboats) by performing an inner join between boats and motors. Of course, this query would miss any motor boats being sold without an engine, and while the query might exclude small sailboats, larger boats (including our 25' Hunter sailboat) generally have motors. So while the assumption may be valid when the application is written, based on the schema and data at that time, baking this type of implicit logic into queries within the application introduces subtle dependencies on the data and schema that are hard to track and maintain.

Clearly, although the schema is nicely normalized from a database perspective, it is not in a very convenient form for the application. What's worse, in order to retrieve the desired information (not to mention making updates such as moving an engine to a different boat) implicit knowledge about the relationships between the tables must be baked into the application. Changes to the schema, for example combining the Engines and BoatEngines tables (a reasonable degree of denormalization a DBA may consider in order to improve performance) causes the application to break in ways that are hard to anticipate and resolve.

Representing Inheritance

Now let's say you want to want to add additional information to the schema. First you make explicit the difference between sailboats and different types of motorboats by adding a "Style" column to the table. Then, for sailboats, you add the type of Keel (Fixed, Swing, or Dagger) and number of sails. For ski boats you add whether it has a ski pylon and/or a tower, and for MotorYachts you want to add whether or not it has

Figure 2: Representing Hierarchy in a Single Sparse "Boats" Table

Boats										
RegNum	**Year**	**Make**	**Style**	**Length**	**KeelType**	**Beam**	**NumSails**	**SkiPylon**	**Tower**	**Flybridge**
WN123AB	1977	Hunter	Sail	25	Fixed	8	3	Null	Null	Null
WN234CD	1999	Calabria	Ski	23	Null	8'7"	Null	Yes	No	Null
WN345EF	1962	Del Mar	Motor	16	Null	6'	Null	Null	Null	Null
WN456GH	1957	Harvey	Motor	13.5	Null	5'10"	Null	Null	Null	Null
WN567IJ	1997	Seadoo	PWC	9	Null	3'10"	Null	Null	Null	Null
WN678KL	1996	Bayliner	Yacht	47	Null	14'11"	Null	Null	Null	Yes
...	...	...	...	...	...	...	...	...	...	...

a flybridge. This becomes challenging with the relational data model because each piece of additional information only applies to a subset of the rows within the Boats table. One way to represent this is by extending the base table ("Boats") with members for each derived type, using Nulls for rows in which they do not apply. For example, this information could be expressed in a single sparse "Boats" table as shown in Figure 2.

As we see, the more properties we add for each derived type, the more the schema of the overall table grows and the more we fill in non-relevant fields with Null values. An alternate way to represent this same information would be to break the additional information for Sailboats, Ski Boats, and Motor Yachts into separate tables, as shown in Figure 3.

This layout avoids the need to add sparse columns to the base table for each property of the derived type, but querying becomes even more complex as each query has to join the additional tables in order to include full information for each of the derived types. For example, to return boat registration, year, make, and associated engine information for all motor boats without a tower, we might write something like the following query:

```
SELECT Boats.RegNum, Boats.Year AS Boat_Year,
 Boats.Make AS Boat_Make, BoatEngine.SerialNumber,
 BoatEngine.Year AS Engine_Year, BoatEngine.Make
 AS Engine_Make, BoatEngine.Model AS Engine_Model,
 BoatEngine.HP
FROM (Boats
 LEFT OUTER JOIN (
   SELECT EngineTypes.Make, BoatEngines.BoatID,
                                  EngineTypes.HP
   FROM EngineTypes
    INNER JOIN (Engines
     INNER JOIN BoatEngines
     ON Engines.SerialNumber =
                        BoatEngines.EngineSerialNum)
     ON EngineTypes.Model = Engines.Model
      AND EngineTypes.Make = Engines.Make
  ) AS BoatEngine
  ON Boats.RegNum = BoatEngine.BoatID )
  LEFT JOIN SkiBoats
       ON Boats.RegNum = SkiBoats.RegNum
WHERE Boats.Style In ("Ski","Motor","PWC","Yacht")
 AND (SkiBoats.Tower=False OR SkiBoats.Tower IS NULL)
```

Note that the Boats table must be joined with the SkiBoats table in order to get the Tower information, and we must account for the NULL value in our predicate for rows which are not Ski boats.

Schema Changes

As the tables grow, the DBA may decide to refactor the schema such

Figure 3: Storing Extended Information in Separate Tables

Boats					
RegNum	**Year**	**Make**	**Style**	**Length**	**Beam**
WN123AB	1977	Hunter	Sail	25	8
WN234CD	1999	Calabria	Ski	23	8'7"
WN345EF	1962	Del Mar	Motor	16	6'
WN456GH	1957	Harvey	Motor	13.5	5'10"
WN567IJ	1997	Seadoo	PWC	9	3'10"
WN678KL	1996	Bayliner	Yacht	47	14'11"
...	...	...	...	...	...

SailBoats		
RegNum	**KeelType**	**NumSails**
WN123AB	Fixed	3
...	...	...

SkiBoats		
RegNum	**SkiPylon**	**Tower**
WN234CD	Yes	No
...	...	...

MotorYachts	
RegNum	**Flybridge**
WN678KL	Yes
...	...

Figure 4: Completely Separate Tables for Each Boat Type

MotorBoats				
RegNum	**Year**	**Make**	**Length**	**Beam**
WN345EF	1962	Del Mar	16	6′
WN456GH	1957	Harvey	13.5	5′10″
...	...	...	...	...

MotorYachts					
RegNum	**Year**	**Make**	**Length**	**Beam**	**Flybridge**
WN678KL	1996	Bayliner	47	14′11″	Yes
...	...	...	...	...	...

PWC				
RegNum	**Year**	**Make**	**Length**	**Beam**
WN567IJ	1997	Seadoo	9	3′10″
...	...	...	...	...

SkiBoats						
RegNum	**Year**	**Make**	**Length**	**Beam**	**SkiPylon**	**Tower**
WN234CD	1999	Calabria	23	8′7″	Yes	No
...	...	...	...	...	...	...

SailBoats						
RegNum	**Year**	**Make**	**Length**	**Beam**	**KeelType**	**NumSails**
WN123AB	1977	Hunter	25	8′	Fixed	3
...	...	...	...	...	...	...

that all of the information for a particular type of boat is in one table, as shown in Figure 4.

This schema optimizes for queries against a single type of boat at the expense of queries across types of boats. And, of course, it means that the queries within your application have to change. Our query for boat and engine information becomes:

```
SELECT Boats.RegNum, Boats.Year AS Boat_Year,
 Boats.Make AS Boat_Make, BoatEngine.SerialNumber,
 BoatEngine.Year AS Engine_Year, BoatEngine.Make
 AS Engine_Make,BoatEngine.Model AS Engine_Model,
 BoatEngine.HP
FROM (
  (SELECT RegNum, Year, Make, Tower FROM SkiBoats
  UNION ALL SELECT RegNum, Year, Make, NULL As
                    Tower FROM MotorBoats
  UNION ALL SELECT RegNum, Year, Make, Null AS
                     Tower FROM PWC
  UNION ALL SELECT RegNum, Year, Make, Null AS
                     Tower FROM MotorYachts
) AS Boats
LEFT OUTER JOIN (
 SELECT EngineTypes.Make, BoatEngines.BoatID,
                         EngineTypes.HP
 FROM EngineTypes
  INNER JOIN (Engines
   INNER JOIN BoatEngines
   ON Engines.SerialNumber =
                   BoatEngines.EngineSerialNum)
   ON EngineTypes.Model = Engines.Model AND
                  EngineTypes.Make = Engines.Make
   ) AS BoatEngine
   ON Boats.RegNum = BoatEngine.BoatID )
WHERE (Boats.Tower=False OR Boats.Tower IS NULL)
```

Note that in order to query across all types of boats including the Tower column specific to SkiBoats, we must explicitly project a Null value for that field for the other tables within the UNION ALL.

ADO.NET Entities

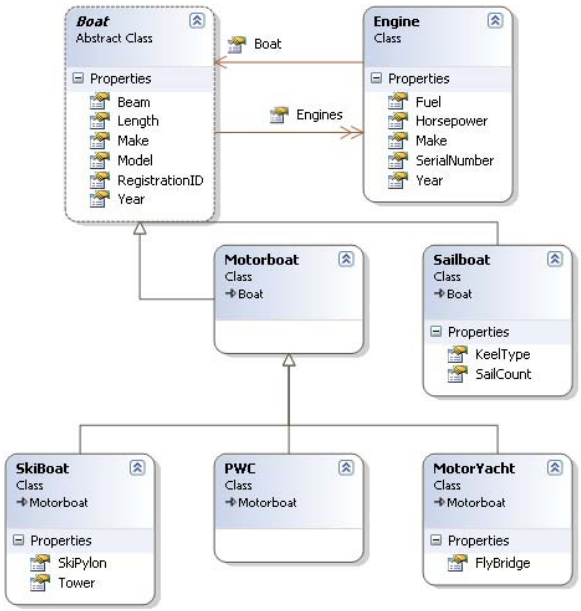
These examples each highlight some of the challenges application developers face today when trying to expose, manipulate, and persist interesting real-world models into a flat relational schema. The fact that the schema may not be owned by the application developer, and may change over time, helps explain why the data access "goo" can occupy such a disproportionate share of an application or framework in terms of code, development, and maintenance cost.

Enter The ADO.NET Entity Framework

The ADO.NET Entity Framework will ship in the first half of 2008 as an extension to the .NET Framework that is part of Microsoft Visual Studio code name "Orcas" and represents the first deliverable in a new Microsoft Entity Data Platform for working with data in terms of a rich, common Entity Data Model. The ADO.NET Entity Framework, is an implementation of Dr. Chen's Entity-Relationship model on top of relational data . Instead of letting storage representation dictate the application model, writing to a common Conceptual Model allows applications greater expressivity in modeling data using real-world concepts which can be flexibly mapped to a variety of storage representations. (For more on the ADO.NET Entity Framework, see Resources.)

The Entity Framework uses a Client View mechanism to expand queries and updates written against the conceptual model into queries against the storage schema. The expanded queries are evaluated entirely within the database; there is no client-side query processing. These Client Views may be compiled into your application for performance, or generated at runtime from mapping metadata provided in terms of XML files, allowing deployed applications to work against different or evolving storage schemas without recompilation.

Figure 5: An Application-Oriented Conceptual Model



An Application Model

Figure 5 shows a more appropriate application-oriented data model for our web application. Note that, although I use a class diagram to represent the model, objects are just one way to expose the conceptual model to the application in The Entity Framework. The same conceptual model could be targeted directly using an extended SQL grammar and returned as polymorphic, hierarchical records.

The first thing we notice about this model is that this doesn't look anything like any of the previous storage schemas. For example:

1) The Engine class contains information from both the EngineTypes table (Make, Horsepower, and Fuel) and the Engines table (Year and SerialNumber).
2) Rather than a BoatID property, the Engine class contains a reference to a Boat.
3) Boats contain a collection of zero or more engines.
4) Rather than expose a Style property on Boat, or have distinct tables for each, we've used the more natural inheritance concept to distinguish the different types of boats.

The Entity Framework allows you to expose this conceptual model to the application, using application concepts such as strong typing, inheritance, and relationships, over any of the previously described database schema. The fact that the mapping is done declaratively, outside of the application, means that if the database schema evolves over time to optimize for different access patterns, only the mapping has to change; the application can continue using the same queries and retrieve the same results against the same conceptual model.

Let's see how working with this conceptual model simplifies our application patterns.

Querying the Conceptual Model

Given this conceptual model, querying a single polymorphic set of boats becomes much simpler. For example, the following code uses this conceptual schema to query the registration year, make, and engine information for all motorboats without a tower.

```
SELECT boat.RegNum, boat.Year, boat.Make,
                              boat.Engines
FROM Boats AS boat
WHERE boat IS OF (Motorboat)
  AND (Boat IS NOT OF (SkiBoat)
       OR TREAT(boat AS SkiBoat).Tower = False)
```

Note that no joins are required in the query; entities are strongly typed, relationships are traversed through properties, and collections can be filtered according to types within the hierarchy.

Nested Results

In each of the first three queries written directly against the database schema, the results would look something like Figure 6. Note that the last boat (the 1996 BayLiner) appears twice. From looking at the data we see that the 1996 Bayliner is a MotorYacht with two Hino 310 engines. Because relational data is flat, there is no good way to represent multiple engines in a single row of the result, so two rows are returned for the same boat; one for each engine.

The query against the conceptual model returns an "Engines" column with a single row for each boat containing the collection of engines as shown in Figure 7.

Alternatively, if only a subset of the data for each engine is desired (for example, the Make and HP) that information can be projected out as follows:

Figure 6: Nested Results Returned as a Rectangular Table

RegNum	Boat_Year	Boat_Make	SerialNum	Engine_Year	Engine_Make	Engine_Model	HP
WN234CD	1999	Calabria	M30099	1999	Mercruiser	350MagMPI	300
WN345EF	1962	Del Mar	M3060	1962	Mercury	Mark30	30
WN456GH	1957	Harvey	T5090	1990	Tohatsu	M50CEPTS	50
WN567IJ	1997	Seadoo	R8596	1997	Rotax	720CC	85
WN678KL	1996	Bayliner	H31096A	1996	Hino	W06DTA	310
WN678KL	1996	Bayliner	H31096B	1996	Hino	W06DTA	310
…	…	…	…	…	…	…	…

Figure 7: Results Returned as a Nested Table

RegNum	Year	Make	Engines
WN234CD	1999	Calabria	SerialNum: M30099; Year: 1999; Make: Mercruiser; Model: 350MagMPI; HP: 300
WN345EF	1962	Del Mar	SerialNum: M3060; Year: 1962; Make: Mercury; Model: Mark30; HP: 30
WN456GH	1957	Harvey	SerialNum: T5090; Year: 1990; Make: Tohatsu; Model: M50CEPTS; HP: 50
WN567IJ	1997	Seadoo	SerialNum: R8596; Year: 1997; Make: Rotax; Model: 720CC; HP: 85
WN678KL	1996	Bayliner	SerialNum: H31096A; Year: 1996; Make: Hino; Model: W06DTA; HP: 310 SerialNum: H31096B; Year: 1996; Make: Hino; Model: W06DTA; HP: 310
...	...	...	...

```
SELECT boat.RegNum, boat.Year, boat.Make,
       SELECT engine.Make, engine.HP
       FROM boat.Engines AS engine
FROM Boats AS boat
WHERE boat IS OF (Motorboat)
 AND (Boat IS NOT OF (SkiBoat)
      OR TREAT(boat AS SkiBoat).Tower = False)
```

Note that this query still does not require the developer to write any joins; the relevant fields from the engine are projected out as a nested column using boat.Engines as the source for the subquery.

Different Views for Different Applications

Web application frameworks in particular often expose different views of the same data through different web applications. For example, the data you expose to unauthenticated web clients may be a subset of the data exposed to preferred members, which may be modeled differently than the data exposed to internal administration and reporting applications. Similarly, the schema of the data you work with within your application framework may differ significantly from the schema of the data you exchange in Business to Business transactions. The ADO.NET Entity Framework facilitates these types of scenarios by allowing multiple conceptual models to be mapped to the same database schema.

Modeling Results as Objects

The previous example shows returning results as records. In the case of the ADO.NET Entity Framework, this means returning results as a DataReader which has been extended to support type information, polymorphism, nesting, and complex values. With Entities it is also possible to write queries against the same conceptual model and return results as strongly typed business objects. When modeling results as business objects, relationships may be navigated and updated through typed properties on the objects rather than manipulating scalar foreign key values. The business objects may optionally be identity resolved and change tracked.

Use of the conceptual model through business objects is illustrated by the following code example. This example shows querying against the conceptual model to return boats as objects, navigate through properties to the collection of engines, and remove any engines that are not the same year as the boat. Changes are saved to the database through the call to SaveChanges().

```
// Specify query as an eSQL string
string eSql =
  "SELECT VALUE boat FROM Boats AS boat " +
  "WHERE EXISTS(" +
    "SELECT engine from boat.Engines AS engine " +
    "WHERE engine.Year != boat.Year)";

BoatInventory inventory = new BoatInventory();
ObjectQuery<Boat> motorizedBoats =
              inventory.CreateQuery<Boat>(eSql);
// Include Engines in results
motorizedBoats.Span.Include("Engines");
// Loop through Engines for each Boat
foreach(Boat boat in motorizedBoats) {
  foreach(Engine engine in boat.Engines) {
    if(engine.Year!= boat.Year)
       boat.Engines.Remove(engine);
       // alternatively
       // engine.Boat = null;
  }
}
inventory.SaveChanges();
```

"THE FACT THAT THE SCHEMA MAY NOT BE OWNED BY THE APPLICATION DEVELOPER, AND MAY CHANGE OVER TIME, HELPS EXPLAIN WHY THE DATA ACCESS "GOO" CAN OCCUPY SUCH A DISPROPORTIONATE SHARE OF AN APPLICATION OR FRAMEWORK IN TERMS OF CODE, DEVELOPMENT, AND MAINTENANCE COST."

Note that, as in the previous conceptual query examples, no joins are required in the query. The object results are strongly typed and updatable, and navigating and modifying the relationship between different types is done through properties and methods, rather than updating scalar key values.

The same query could be written using the new Language Integrated Query ("LINQ") extensions being introduced in the next version of Microsoft Visual Studio and the .NET Framework (codenamed "Orcas") as shown below:

```
BoatInventory inventory = new BoatInventory();
 // Include Engines in queries for Boats
 inventory.Boats.Span.Include("Engines");

 // Specify query through LINQ
 var motorizedBoats =
   from boat in inventory.Boats
   where boat.Engines.Any(e => e.Year != boat.Year)
   select boat;

 // Loop through Engines for each Boat
 foreach(Boat boat in motorizedBoats) {
   foreach(Engine engine in boat.Engines) {
     if(engine.Year != boat.Year)
       boat.Engines.Remove(engine);
       // alternatively
       // engine.Boat = null;
   }
 }
 inventory.SaveChanges();
```

Conclusions

In summary, there are at least six reasons why writing applications directly to the database storage schema may be problematic:

1) You may not have control over either the application object model or the storage schema.
2) The degree of database schema normalization may make it cumbersome to consume directly from an application.
3) Certain real-world modeling concepts can't be directly represented in a relational schema.
4) The application may be forced to embed implicit knowledge of how fields in the database schema are used that is difficult to track and brittle to maintain.
5) Different applications may want to expose different views of the same data.
6) The database schema may change over time, breaking applications that directly write to that schema.

The ADO.NET Entity Framework allows your applications to target a conceptual model using application concepts such as strong typing, inheritance, and relationships. This conceptual model can be mapped to a variety of storage schemas. Because the mapping is done declaratively, outside of the application, changes to the database schema over time to optimize for different access patterns only requires that the mapping change; the application can continue using the same queries, retrieve the same results, and make changes against the same conceptual model.

Resources

Next-Generation Data Access: Making the Conceptual Level Real, J. Blakeley, D. Campbell, J. Gray, S. Muralidhar, and A. Nori (MSDN, June 2006)
http://msdn2.microsoft.com/en-us/library/aa730866(VS.80).aspx

The ADO.NET Entity Framework Overview (MSDN, June 2006)
http://msdn2.microsoft.com/en-us/library/aa697427(VS.80).aspx

The LINQ Project (MSDN, January 2007)
http://msdn2.microsoft.com/en-us/netframework/aa904594.aspx

Visual Studio Future Versions (MSDN, January 2007)
http://msdn2.microsoft.com/en-us/vstudio/aa700830.aspx

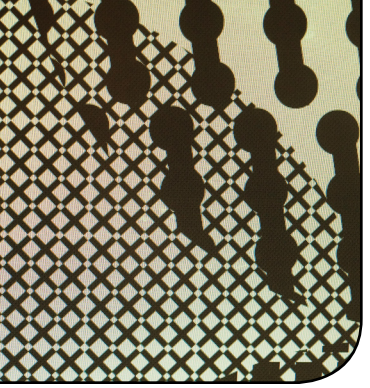
Dr. Peter Chen: Entity Relationship Model - Past, Present and Future, April 2007
http://channel9.msdn.com/Showpost.aspx?postid=298587

The Entity-Relationship Model — Toward a Unified View of Data, P.P.S. Chen. ACM Transactions on Database Systems (TODS), 1976.

A Relational Model of Data for Large Shared Data Banks, E. Codd. Communications of the ACM, 1970.

About the Author

Michael Pizzo has worked for over 17 years in the design and delivery of data access solutions and APIs at Microsoft. Michael got started in data access as a program manager for Microsoft Excel in 1989, and was involved in the design and delivery of ODBC, along with the ODBC-based Microsoft Query Tool shipped with Microsoft Office. He has been active in the standards organizations, sitting as Chair for the SQL Access Group, working with X/Open on the CAE specification for "Data Management: SQL-Call Level Interface (CLI)", serving as Microsoft's representative to the ANSI X3H2 Database Committee, and as an elected ANSI representative to the ISO committee meetings that defined and adopted Part 3 of the ANSI/ISO SQL specification for a call-Level Interface (SQL/CLI). Michael was a key designer and driver of Microsoft's OLE DB API, and later owned the design and delivery of ADO.NET version 1.0. He is currently a software architect in the Data Programmability Team at Microsoft, contributing to the architecture and design of the next version of ADO.NET and core building block for Microsoft's new Entity Data Platform, the ADO.NET Entity Framework.



Architecture Journal Profile: Pat Helland

Recently, the Architecture Journal stopped by the office of Microsoft Architect Pat Helland to catch up with him and ask him about his thoughts on architecture, and how to become an architect.

AJ: Who are you, and where do you work?

PH: My name is Pat Helland and I work in the Visual Studio team in the Developer Division. I came to Microsoft in 1994 and helped found the team that built Microsoft Transaction Server and the Distributed Transaction Coordinator. By about 1998, I started to understand that N-tier computing was not addressing all the issues our customers were having and I became involved in what today is called service-oriented architecture, which back in the day I was calling "autonomous computing." This led to working on connecting the services with reliable messaging, and I helped start a project which shipped in SQL Server 2005 called SQL Service Broker. Starting in 2003, I spent some time working in DPE doing evangelism to our enterprise customers. [See sidebar for more on Pat's long career and varied projects.]

AJ: As an architect for many years, what kind of advice would you give to someone who wants to become an architect?

PH: Architecture is a very interesting area. I liken it to building architecture. If you stand back and think about what a building architect has to do, they first of all have to think about a business need and understand how to create a building that fulfills that business need.

Even more, the building needs have a feel to it. When someone walks up and looks at the building, some emotion is conveyed.

At the same time, we have to deal with a myriad of pragmatics. How is the air going to flow through the building? How will the people move through the elevators? How do you make it comfortable and meet all other environmental considerations? You have the functioning of the building—the utilitarian object—combined with the fulfillment of the business needs, combined with the effect that particular structure will have on human beings.

Looking at that from the standpoint of software, you see exact parallels. The primary goal of a software architect is to understand and work out how to fulfill the business need. At the same time, you want the software to relate to people and their feelings about using it. While you are doing that, you have to deal with all of the transcendent aspects of the darn thing functioning correctly and pragmatically.

A building architect dealing with a large building may not be an absolute expert on elevators or airflow. But he has to know enough about those areas to interact with the experts and bring it together into a cohesive whole. Similarly, a software architect needs to know enough about the different aspects of the larger system that they are putting together to interact with the respective specialists.

AJ: A lot of this may sound familiar to people who have heard about Metropolis, correct? [See Resources.]

PH: Yes, and I'm still very keen on the Metropolis theme. Take a look at the history of urban development and how cities changed when the railroads arrived in the middle of the nineteenth century. The rapid movement of goods and people allowed—and caused—the same changes to occur in cities that we are now seeing in IT shops as different computers and facilities are being interconnected to the Internet.

The ability to communicate and to carry data or carry requests for business functionality remotely is causing pressure for standardization. It's causing the work to be broken into pieces and then connected with standards. Those standards are not just about how to squirt the data back and forth, but also to answer questions: What is a service? How do I think about the operation that I want to provide?

Again, this is very much what we saw in cities when the manufacturer was separated from the retailer who was delivering the goods. You can't do that unless you standardize on SKUs [Stock Keeping Units]. When you talk about breaking manufacturing down, and having manufacturers building parts for larger things, you are talking about standardization.
You see a lot of commonality when you look at cities and when you look at the development of widely distributed IT shops.

AJ: I totally agree. Working in the developer division, I'm sure you need to stay up-to-date with the latest technologies. How do you do that?

PH: For me, I'm a relatively social person and I find time to talk both to people who are experts in the area—ask them their perspective—and compare their views to other experts that I talk to. In the end you just have to read. You have to think. It's actually a huge challenge trying to balance the time when you are talking and interacting with people where you are gaining and helping in so many different ways, and the time to go away to contemplate. Again to read, to absorb, to let things cook in the background and try to figure out what kind of big, sweeping changes you can make. And then you have to go and try to popularize those. It's astonishing how much of the job of an architect is to evangelize, to socialize—to get people to buy into a dream. Sometimes people ask what my job is, and I tell them, "To make stuff up and sucker people into building it." That's the job of an architect!

AJ: I like the definition. You mention socializing and keeping up-to-date with people. Who is the most important person you have ever met—and why?

PH: Important within the context of the computer field? I would say my dear friend Jim Gray whom I've known since 1982 and has recently gone missing, which is quite a hardship on many people. The reason I love him and hold him up as a mentor is that he cares compassionately about individuals. He doesn't care if the individual was some important muckety-muck or an aspiring college kid; he just cares about them as individuals. Related to that, Jim has an extraordinary ability to look at someone, listen to them for a minute and rapidly gravitate to what level they can understand the discussion. He neither talks down to them, nor overwhelms them with technology and concepts that are out of their grasp. And I've watched him do it repeatedly. I've watched him do it for 25 years, and I've always been just very, very respectful of that.

AJ: Yes that is an amazing thing, a very great man.

PH: This has allowed him to build an enormous network. I've watched Jim see someone in need of something, and then do a little bit of matchmaking with someone else. The combination that is then created can be very powerful and results in wonderful technology and wonderful growth in the technology community. Jim is constantly working to help uplift other people, each time concurrently looking at how he can uplift the technology and the business. It's that combination of growing individuals while accomplishing the business goals of creating great software that is such a joy and that has earned Jim so much respect in the industry.

AJ: Looking back at your career, what do you most regret?

PH: Not following Jim's advice and writing more. I am not always as disciplined as I wish I were in terms of forcing myself to write down all the things that are on my mind. Taking the ideas that are rattling around in the cavernous emptiness above my shoulders and getting them written down is something that I'm again anxious to push forward. Jim has always told me: You need to write more; you need to communicate more.

I don't know what it's like for everyone else, but I know it's hard for me to take the time and create a cohesive, communicative document unless I create a deadline and unless I get myself under the pressure. Then, I have to go away for a few days isolate myself with "Oh, my gosh! I'm going to get this done or bad stuff is going to happen!" I have to force myself to finish it, tie a ribbon around it, and push it out even though the perfectionist in me wants to rewrite it. That is something I wish I could've done better and I continue to try to make happen more.

AJ: The forcing function, I think a lot of people struggle with that.

PH: Yes. Architects are human. You just have to figure out how to make it happen.

AJ: So, what do the next few years look like? What do you hope to accomplish as part of your job?

PH: My goals are to figure out how to pick up a certain area of product features and drive them to a new level. Now exactly what it's going to be, I've only been back like five weeks and so I haven't completely put shape and form to that, but I want to push some new cool stuff into our product sets.

At the same time, I would like to gradually increase my time out presenting, my blogging, writing the artifacts that help people understand the various things I've been thinking about.

Also, just working with people inside and outside of Microsoft to help them solve their problems is what I find joyous—to really make a difference. I feel Microsoft is special in that you can have a broad influence.

Now mind you, it's an interesting place with a lot of stuff going on and lots of chaos here and there. Sometimes the things you are working on don't make it and not everyone realizes the incredible emotions that are tied up in engineering. People invest themselves in stuff which may or may not stick. I'm interested in how to facilitate avoiding the pain in that and how to bring out the joy. I'm also interested in helping people out into the public with their ideas. This is what I strive for.

Pat Helland
Microsoft

Pat Helland has almost 30 years experience in scalable transaction and database systems.

In 1978, Pat worked at BTI Computer Systems where he built an implementation language, parser generator, Btree subsystem, transaction recovery, and ISAM (Indexed Sequential Access Method).

Starting in 1982, Pat was chief architect and senior implementor for TMF (Transaction Monitoring Facility) which implemented the database logging/recovery and distributed transactions for Tandem's NonStop Guardian system. This provided scalable and highly-available access to data with a fault-tolerant message based system including distributed transactions and replication for datacenter failures.

In 1991, Pat moved to HaL Computer Systems where he discovered he could work on hardware architecture. He drove the design and implementation of a 64-MMU (Memory Management Unit) and a CC-NUMA (Cache Coherent Non-Uniform Memory Architecture) multi-processor.

By 1994, Microsoft called and asked Pat to work on building middleware for the enterprise. He came and drove the architecture for MS-DTC (Distributed Transaction Coordinator) and MTS (Microsoft Transaction Server). Later, Pat became interested in what today is called SOA (Service Oriented Architecture) and started a project to provide high-performance exactly-once-in-order messaging deeply integrated with the SQL database. This shipped in SQL Server 2005 as SQL Service Broker. Later, Pat worked on WinFS and did a stint in DPE evangelizing to our largest enterprise customers.

For the past two years, Pat had been working at Amazon on the catalog, buyability, search, and browse areas. He also started and ran a weekly internal seminar series. He is excited to return home to Microsoft and to work in the Visual Studio team!

Resources

Metropolis
http://www.pathelland.com/presentation_overview/index.htm

Data Confidence Over the Web

by Peter Hammond

Summary

The successful exchange of data is the most integral function of applications today, and yesterday, for that matter. This article highlights some experiences from the viewpoints of an end user, novice developer, professional consultant, and the development team lead over the course of the past 10 years. I've had some fantastic successes and disastrous failures, and gained many insights along the way. I detail how the architecture of the past combined with the technology of today has led to successful exchange of data over the Web.

In the fall of 1994, I was transferred to Fairchild Air Force Base in Washington State after a four-year tour at the beautiful Aviano Air Base in Italy. One of my responsibilities as a security police desk sergeant—a cross between a police dispatch and 911 operator—was to document the activities during my shift. In Aviano, this consisted of a typed 12+ page report with 10 carbon copies, each one distributed to the command for daily review. If any part of the report had to be fixed, the entire report had to be retyped and carbon copies redistributed. We killed a lot of trees. You can imagine my excitement at Fairchild when I was introduced to the new Windows-based Security Police Automation System (SPAS) for police reports. I would later learn that acronyms can predict trouble down the line. This networked client/server application built on FoxPro was a wondrous thing, as you could type out all your reports, correct any issues found, and distribute to command digitally. No more trees would die in vain. So it seemed.

Unfortunately, as time passed and the amount of data grew, SPAS freaked out. After spending hours typing out the detailed notes on an incident and clicking Save, a cryptic "Index Corrupted" error would flash. Even though your work still appeared on the screen, in reality it was gone, although you might not know it until command pulled you back in after you had gone home because your report was missing.

The consequences were swift. Everyone started doubting the SPAS. Superstitions arose about what would cause the system to fail. Was it hitting Save more than once, having a large amount of text, or standing on one leg while you typed with one eye closed? No one knew for sure. In the end we started printing out the entire report again, just in case. Thus began my understanding of the importance of having confidence in data systems. Without confidence in solid data communications, the users (and trees) pay a terrible price.

I started to create my reports in Microsoft Word and then cut-and-paste them into SPAS. I was getting my work done faster since I didn't have to retype all of my entries, even if there was an "Index Corrupted" error, and I quickly realized I had free time. The value of having technology work for me rather than the reverse wasn't lost on me. I spent my newfound time learning more about Microsoft's Office products.

As luck would have it, I got the opportunity to start doing light development building computer-based training programs in Microsoft PowerPoint for the Air Base Operations course (ABO). I created presentations with embedded graphics, sound, and video which made the training a big hit. When we needed to track who had completed the training, instead of using a paper signature sheet, I wrote a testing program that would track who had attended the training. Microsoft Excel was my first choice because that seemed to be where everyone tracked data, from flu shots to training records to test scores, and so on. Using Excel, I was able to define the questions as one sheet and the answers in another, quickly build a testing form, and successfully demonstrate it to my supervisor.

All seemed well until some obvious issues arose. We posted the Excel test to a network share to enable multiple people to take the test only to find out it could only be opened by one person at a time for edit and read-only mode for everyone else. I used hidden sheets to work around the security issue of storing the questions and answers in the same file. These measures failed as people found it faster to use the paper test forms rather than to wait for the single file to become available and some users discovered how to unhide the answer sheet. Yes, more trees suffered.

Little did I know that my baptism by fire into the difficulties of using network access for multi-user tenancy and ensuring real data security was just the tip of the iceberg.

I found Microsoft Access which promised to solve these problems and more, with its support for multiple simultaneous users, user-based security, plus the ease of Excel in defining tables, the ability to have complex data schema, and the advanced form designer.

The new Access testing database was a great success and I was granted server space on the base's CD tower server in the network center so the entire base could complete the mandatory training and tests remotely. The successful results quickly poured in and the training appeared to be going very well—unfortunately, a little too well.

I happened to notice that everyone was getting a 100 percent on all the tests the first time. During the design, I had included a creation and modification timestamp on each user's answer. I knew something was amiss when I ran a report using the difference between the created and modified timestamps on the answers. Most users were finishing the 30 question test in less than 27 seconds. You can imagine my surprise, as I figured out they were cheating! As the test questions

were the same sequence, someone had created a cheat template (on paper, I'm sure). Serving the questions randomly helped ensure the tests were being taken legitimately. However, now that people weren't cheating, I had a new problem: The tests were taking longer to complete, so more people were accessing the tests at the same time. The Access database brought me to the harsh realization of what (lack of) scalability meant.

Apparently, the CD server started throwing a "Record Lock Timeout Error." Each one resulted in a notification bell that got pretty annoying to the network administrators, who weren't very happy that I (a security policemen) was playing developer. So they shut off the bell. As the number of users grew and the number of record-locking timeout errors increased, the inevitable occurred: My little Access database dropped the CD tower server and lost multiple in-process tests, including those of several very unforgiving officers. This was particularly bad on its own, however, the impact was far-reaching as the base was in the middle of a huge deployment of both Office 95 and Windows 95 from the CD server. I solved the issue by limiting the number of simultaneous users; however, the lessons of multiuser concurrency, data integrity, network resources and scale were not wasted on me.

In the end, the ABO course was a big success and the training was well received. I went on to build several more database programs with Access, enjoying its rapid development features but always wary of its limitations.

WORMS, ARROW, and SQL Server

After I completed my second enlistment, I decided to try the civilian sector as a professional consultant. I found a lot of work fixing and upgrading a myriad of Excel- and Access-based systems. It seemed the world ran on Excel, and not very well. I eventually migrated into Visual Basic which enabled me to use Access as a back end and code around most of its issues.

I was afforded the opportunity to work on the Work Order Remote Management System (WORMS). WORMS was based on Microsoft SQL Server 6.5 and was an incredible system. Its scale was huge as it serviced hundreds of users, processing thousands of records. But the system had gone to the worms.

Apparently, WORMS was losing records at random. The loss occurred somewhere between the local SQL database and an Oracle system of the parent company. As a stopgap measure, all the work orders were printed out on paper in the morning, then hand-entered into a separate Access database that was copied to corporate each day to reconcile lost records. You guessed it, no data confidence and more trees killed in a single day then I did in my entire military career.

My job was to babysit the system while its replacement, the Automatic Remote Routing Of Workorders (ARROW) was developed. As I tracked down the loss of the records, I was very impressed with the diagnostic capabilities in SQL—from tracing to event logs, I could track all the data from end to end. With this advanced level of diagnostics, I was able to determine the cause and fix it, subsequently finding the records and losing my job all in one fell swoop.

The fix worked so well that it drew into question whether the customer needed the replacement ARROW system and I was quickly pulled from the project to ensure I didn't "fix" anything else. Regardless of the outcome, I was hooked. SQL Server was the answer I had been looking for, and then the real questions started.

As I advanced with SQL Server, the successes followed. Client/server-based applications were solid, easy to build, fast to troubleshoot, and users loved them because when they hit Save, it saved. Confidence in data equaled successful applications.

Years later, I was contracted to work on a project for tracking training that Microsoft conducted as an intranet academy. Using Microsoft .NET and SQL Server 2000, we quickly built a client/server application that met the users' needs extremely well. Over time, however, the number of support issues increased markedly as people tried to use the application remotely. Since it had to load all the records from the server each time it was launched, the performance over a VPN connection was dreadful. Nevertheless, users were happy because they had confidence that their changes were being persisted immediately, and with all the records loaded in memory, the UI was very responsive. The application was used for years.

Eventually, the resource requirements, network connectivity issues, and the explosion of the Internet caused the popularity of client/server architecture to fade. N-Tier implementations, with Web services, were hailed to be the holy grail of networked applications for online data exchange.

Web Services and CyberSavvy

We jumped on board without looking back. I say "we" as I had started CyberSavvy, a vending company providing high-end consulting services. I hired the best of the best, some that I had had the privilege of working with over the years or by referral only. We focused on creating best-of-breed solutions for Microsoft's toughest internal business problems.

Web applications were the buzz, but Web interfaces that lost all your data when the page errored or connection went down did not build data confidence. Instead, we focused on building smart client applications using Microsoft .NET. They provided the rich features needed for applications that addressed complex business problems, involving complex relational data from many sources, requiring instant results and the ability to work both on- and offline.

Over the last five years, we built 15 smart client applications for Microsoft and built a strong working relationship with Microsoft's ClickOnce, .NET, ADO, and SQL teams. Most of these applications focused on solving issues facing Microsoft's business and sales force.

The Enterprise Product Roadmap

One of the most interesting solutions we built for Microsoft helps keep the field updated with new product release information.

Since the sales force often works in disconnected environments, either in transit or at a customer's site, the necessity for offline ability prohibited a Web application or typical client/server solution.

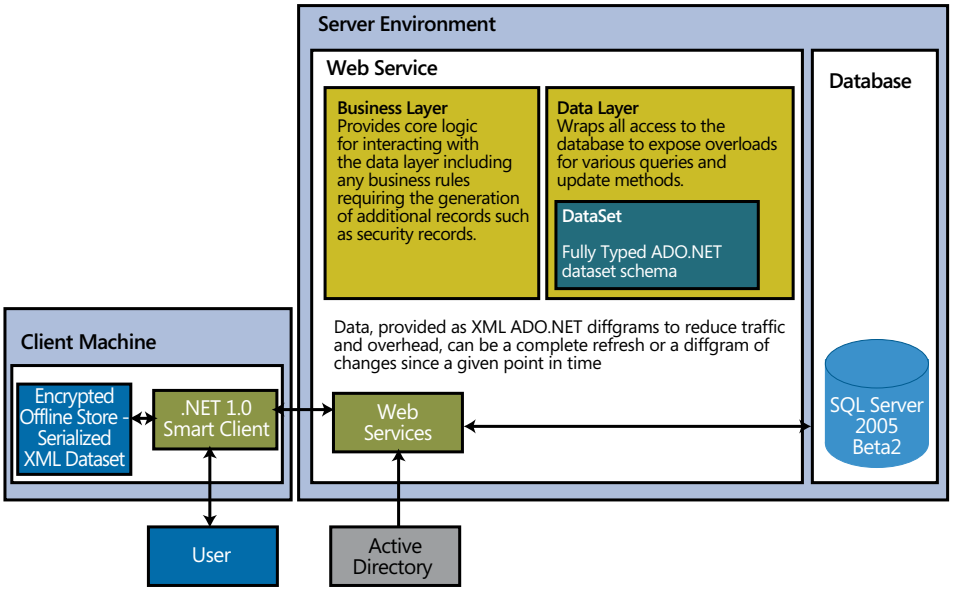
The requirements to quickly search, sort, and filter through all Microsoft products for custom presentations necessitated having all the data locally, yet kept up-to-date. The added need for intense security as certain product information had to be filtered by granular roles required row-based permissions.

By using SQL Server, Web services, and a smart client distributed with ClickOnce, we achieved fantastic results in all sectors. (See Figure 1.)

The Enterprise Product Roadmap (EPR, not to be confused with ERP) was rapidly adopted by the field, scaled incredibly well, was secure, worked online/offline, and was highlighted as a Microsoft showcase.

One disadvantage was trying to create and maintain a Web service

Figure 1: Original Architecture for the Enterprise Product Roadmap as of .NET 2.0 Beta 1



design that used multiple Web methods to select and post data to each individual table (EPR had 20+ tables). We standardized on automated Select, Insert, Update, and Delete stored procedures that were used to create a dynamic ADO.NET Dataset DiffGram and serialized to XML a Web service that had just two methods.

The select Web method queried all of the data from all of the EPR tables a user had access to and returned a server datetime that annotated when the data was queried. Subsequent use by the same user would result in a DiffGram dataset, containing XML that represented new or updated data as well as deletes.

The post Web method took a DiffGram with pending changes. These changes were then sorted by their change type and foreign key relationships so that they may be committed to the SQL database in the correct sequence. That way we had Web methods that did not require changing if we modified our schema. Nevertheless, as the schema was revised, we still had to revise the datalayer that these two Web methods relied upon. For each new column or table, stored procedures, dataSet schema, and the Web service's datalayer code needed to be revised to match; a process that we automated over time. This model allowed for rapid updates to the smart client's data cache by using ADO.NET to do all the hard work.

We had extraordinary success with this on many applications—that is, until we tried it with the Emerging Business Team Tracker (EBT).

The Emerging Business Team Tracker

With many successful implementations of this architecture behind us, we tackled EBT with great confidence. We had created automation processes to dynamically build out the data and Web service layers from our standardized stored procedures. Our security fully integrated with Active Directory with the ability to filter user data at the row level. We built robust client applications using WinForms with all the bells and whistles of highly polished applications that users raved about. Our implementations were considered Microsoft showcases and used by the field to help sell the power of Microsoft technologies. Regardless of our previous successes, EBT would soon make an important point painfully apparent. Looking back, it's clear that the majority of our projects were report and query applications, basically read-only. EBT was a horse of a different color, and it was about to remind me of an old lesson.

EBT was both a reporting and a data capture program, comprising three separate systems that we combined into a single solution. The promise of a single application providing offline access and advanced search was greatly anticipated by the users. We released the application to great fanfare, confident of the success of the solution. We were completely unaware of the horrors to come.

At first everything went very well. Most users were connected to the corporate infrastructure while using EBT and making data changes. The amount of changes in a single DiffGram posted to the Web service was very small. As things progressed, more users worked offline for longer periods of time, and soon the support issues started.

As with all of our other applications, EBT stored its data using in-memory ADO.NET datasets. This worked relatively well for small amounts of data. However EBT's dataset of about 60+mb, when serialized to disk, caused a slow launch time as we had to deserialize all the data into memory on load. The problems really started when we serialized it back out to disk on exit.

We utilized DiffGrams on the client in the same way as the server to communicate pending changes back and forth. Since pending change DiffGrams were persisted on exit, if the operating system or applications caused a fault before it was able to finish, the saved changes were lost. Sometimes the crash happened after the UI closed, so the user wouldn't know there was even an issue until the next time they launched and noticed their changes missing. We tried to compensate for this by doing incremental saves while the application was running but it caused various other issues, such as lags in the performance. Regardless of how well we did it, it wouldn't handle 100 percent of the cases.

Multiuser concurrency.

The next big issue was multiuser concurrency. As the DiffGrams were basically sequential, Insert/Update/Delete actions, if any part of the steps failed when applied to the server, the entire DiffGram failed. This manifested itself when a user working offline for several days making huge amounts of changes would synchronize and one of their first changes would be rejected due to a concurrency violation because the row had been updated by someone else while they were offline.

That caused the subsequent actions to all fail back to the client. We tried to resolve the issue automatically but the client side was usually so out of sync with the server, the user's changes compounded in a

series of conflicts which currupted the local DiffGram. The user would then force a "Full Data Reset" option, deleting their cache and pulling down a new copy. Of course, this caused all their pending changes to be lost as well. We built more and more diagnostics into the calls and more and more failover retry logic, but users lost confidence that their work would really be saved.

The worst was yet to come

The applications usage and the amount of changes being applied by both the server and the client rose exponentially. The amount of Insert/Update/Delete changes that needed to be reconciled at the same time by ADO.NET started to have some all too familiar issues. You can imagine my horror when a user sent in a support issue with screen shot of a huge error dialog on EBT with the accursed "Index Corrupted". I couldn't believe that I was responsible for recreating SPAS.

"AS OUR USER'S CONFIDENCE IN THE DATA'S INTEGRITY INCREASED SO DID THEIR DEMANDS ON OUR ARCHITECTURE. TO OUR GREAT RELIEF, SQL HAS STEPPED UP TO MEET EVERY NEW CHALLENGE WITHOUT COMPROMISE."

For almost a year, I funded an effort to fix the flaws and make EBT work—from creating asynchronous dataset backups, to identifying several obtuse ADO.NET bugs with large scale DiffGrams, to building advanced logging diagnostics into almost every step of the Web service.

We started exploring memory maps and binary datasets. We wrote an administrative utility that could reload the various individual DiffGrams from their XML files and recreate the entire change set in memory. Then we would try and work through the dataset changes row by row sending them one by one to the Web service to find out which one actually violated a constraint, errored on a security issue, or simply did not work for some indeterminate reason. Once identified, the change(s) could be removed and then an attempt was made to resubmit the rest of the DiffGram. This would often fail as the changes were interdependent due to the complex structure of the data and security.

In the end most attempts were futile—the hierarchical nature of the errors combined with the offline aspect of the changes made it next to impossible to diagnose the original cause of the issues without capturing the state of both the client and the server at the same time.

We later found the same issues were actually occurring with our previous applications as well, such as EPR. But since they used read-only data, when a concurrency issue occurred, a new dataset would automatically be applied over the one in error and the user was unaware that anything had happened. With EBT, the user experience was catastrophic as the new dataset would erase any pending changes they may have had. Support incidents were through the roof, and most of the time the issue was too far gone to even try to fix. Users were literally cursing us, and worst of all, they started saving screenshots of their data edits and then using these to reenter their changes when they were lost. Not quite as bad as printing it to paper but just as damaging to the success of the system.

The more things we fixed and workarounds we applied, the more obvious it was that we were creating custom processes to handle data to Web service synchronization, which we had chosen not to do in favor of using the out-of-the-box ADO.NET DiffGrams. We also couldn't focus on building the remaining application features required by the client which hampered user adoption even further.

SQL Merge Replication

Searching for answers, we engaged the ADO.NET team and did extensive reviews of the architecture. By all rights we had done exactly as prescribed. The basic elements of the architecture worked for smaller implementations or those that were read-only. However, using ADO.NET with DiffGrams to communicate these changes was determined to be inadvisable due to the combination of the size and complexity of the data, the row level security requirements, and the offline model with data entry.

As we searched for new ways to resolve the issues encountered, it became apparent that we needed a client-side database engine to handle the data locally and would need to create a solid middle tier to ensure reliable communication.

We looked into SQL replication seeing that it was essentially what we were doing. Replication had always been represented to me as a great idea, but a terrible reality. We engaged with the SQL team and joined the Yukon TAP program. SQL Merge Replication to synchronize from the SQL Server to a client SQLExpress instance seemed optimal, but would it really work?

Over the course of two months, we completely removed the various tiers of datalayer, data transformations, and Web services from the server and client application. We then rebuilt the EBT client to work directly against the local SQLExpress instance and it depended solely on SQL Server Replication for the data transport.

Client/server simplicity, multiuser tenancy, true scale and offline ability

Success was immediate. The user's changes were saved directly to a local SQLExpress database bringing the same reliability to the client that the full version of SQL brings to the server. The reduction in multiple tiers of custom logic, and the addition of SQL's robust diagnostics and event logging, enabled us to provide quick resolution to almost every data support issue we received. As our users' confidence in the data's integrity increased so did their demands on our architecture. To our great relief, SQL has stepped up to meet every new challenge without compromise.

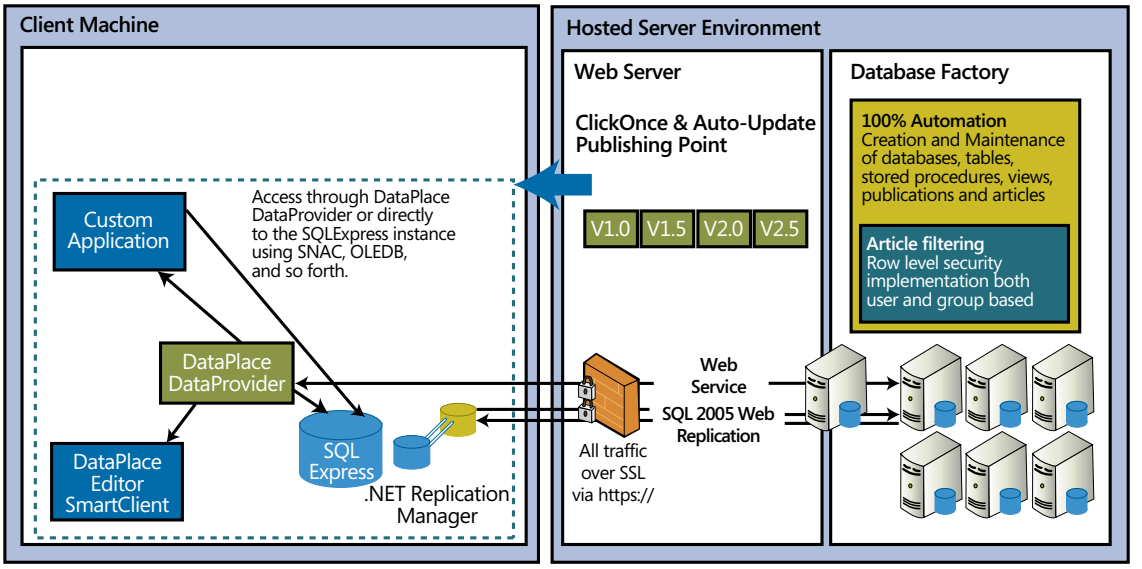
From EBT user on 12/23/2005:

"Sorry but this app has little trust worthiness at this point, and wondering if I should dump everything I do to Word documents."

From same EBT user, five months later, on 5/1/2006 after the new release using SQL replication:

"So far so good – I've done almost every combination of infrastructure usage that would typically create problems for replication or stability and have not had one problem – congratulations!"

Figure 2: DataPlace Architecture



Data Confidence

We continue to find advantages with the new design. For example, replacing ADO.NET in-memory datasets with a true query engine allowed us to strip almost all the extremely complicated data processing code that was necessary to represent the various "views" of the data the customer wanted with standard SQL. Thus, we were able to leverage the DBA for client functionality and to remove a lot of hard to maintain custom code. In some cases, code size dropped by more than one-third.

As we progressed with SQL replication, securely transferring data over the intranet and Internet was a concern. One of the promises of Web services was the idea that the design could be used to communicate data over the Web. Again SQL replication provided the answer, as it came with an out-of-the box implementation of the Web replication component. By using the REPLISAPI component, schema, data and updates to both can be securely and reliably relayed over https.

We have had such great success with SQL replication that we are now providing a Software as a Service (SaaS) product line called DataPlace (see Figure 2). DataPlace exposes the tremendous power of SQL Server and replication for the end users, novice developers, professional consultants, and a development teams, without the complexities of having to set it up for themselves.

It provides a database factory to dynamically design, develop, deploy, and use the power of SQL databases in a fully hosted environment. It is the solution we now use to build all of our latest smart client applications.

This article recounts our experiences over the course of several years and how we've spent a considerable amount of time and effort trying to solve the complex problems relating to working with data over the web in a multiuser, secure, scalable, confident and repeatable way. These experiences are not uncommon among developers, and they illustrate the necessity for dependable products and the knowledge to use them. Microsoft SQL Server 2005 Replication has provided a solid answer to these complex technical problems. Now my team and I can finally focus on what we started out to solve, our customers' business problems!

References

Aviano Air Base, Italy
http://www.aviano.af.mil/

Fairchild AirForce Base
http://www.wafair.ang.af.mil/

Configuring and Maintaining Replication
http://msdn2.microsoft.com/en-us/library/ms151247.aspx

Microsoft.NET
http://msdn2.microsoft.com/en-us/netframework/aa663309.aspx

Smart Client Community
http://msdn.microsoft.com/smartclient/community/scfaq/default.aspx

Microsoft Case Study Smart Client Sales Application Saves Time and Cuts Deployment Costs
http://www.microsoft.com/casestudies/casestudy.aspx?casestudyid=49078

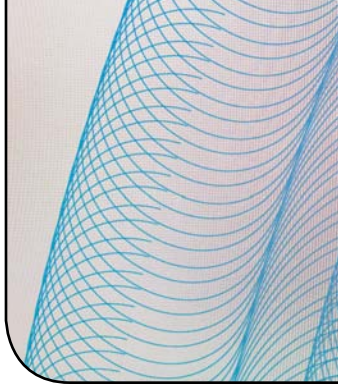
CyberSavvy.NET DataPlace (information and free trial)
http://www.cybersavvy.net/dataplace

About the Author

Peter Hammond is president of CyberSavvy, a Microsoft Preferred Vendor. He served two enlistments in the U.S. Air Force, during which he became interested in programming. After duty, he gained real-world experience working in various roles from network support to development consulting at several companies including Microsoft. In 2002, Peter recruited a team of exceptional professionals focused on building showcase solutions with Microsoft's latest technologies. CyberSavvy has built more than a dozen enterprise applications, notably the Enterprise Product Roadmap, used by more than 10,000 Microsoft employees for more than five years. Peter can be contacted at peter@cybersavvy.biz.

Managed Add-Ins: Advanced Versioning and Reliable Hosting

by Jesse Kaplan



Summary

A new architecture and features for a new managed add-in model will be introduced in the Microsoft .NET Framework 3.5, which will ship concurrently with Visual Studio "Orcas." The next version of the Framework continues our strategy of shipping newer, additive versions at a faster pace than we ship full revisions of the underlying runtime. As 3.0 was a set of additional assemblies on top of 2.0, 3.5 will be a set of new assemblies on top of 3.0.

Solution Architects and application developers wanting to add extensibility face a common set of technical hurdles. The System.AddIn assemblies that are introduced in this next release aim to address two classes of problems that arise when adding extensibility. In this article, we detail how the new managed add-in model and assemblies address each problem set. We'll then describe some advanced hosting techniques that can be used to ensure resiliency and reliability in the face of misbehaving add-ins.

Solution Architects and application developers are typically interested in extensibility for several reasons. One common reason is that their customers each have very specific feature requests that the application itself could never keep up with, so they offer extensibility to let the customers fill in the needs themselves or buy them from a third party. Another common reason for offering extensibility is to build a rich community ecosystem around the product. By offering an extensible application you can engage the community with the product by offering the opportunity to contribute to it in the form of these extensions. Regardless of the reasons for wanting extensibility, once application developers go down this path, they'll face a common set of technical hurdles.

Starting with the next release of the .NET Framework, Microsoft is introducing the System.AddIn assemblies to address two classes of problems that many customers are hitting when they try to add extensibility to their application—we call these the "Version 1" and "V.Next" problems. Version 1 problems refer to set of issues a developer runs into when first adding extensibility to an application: This includes common tasks such discovering, activating, isolating, and sandboxing the add-ins. The V.Next problems on the other hand refer to the set of issues the developer faces as the application changes: keeping old add-ins working on new hosts, getting new add-ins to run on old hosts, and even taking add-ins built for one host and running them on a different host. The new managed add-in model, and the architecture it defines, addresses these V.Next problems, and the features in our new System.AddIn assemblies make solving the Version 1 problems much easier.

Managed Add-In Model Architecture and Versioning

Once an application has shipped (actually, this often happens before it has shipped), developers typically start mapping out all the changes, improvements, and additions they want to make to the application in the next version. The problem developers of extensible applications have to face is that they need to figure out how to do these things while still keeping all the add-ins written against previous host versions working on this new version.

Today this is typically done, in the best cases, by defining new interfaces that inherit from the old ones and have the new host and new add-ins try/casting to see if the objects they are getting back actually inherit from new interfaces or just the old ones. The problem is that this forces both the host and the add-ins to be intimately aware of the fact that these interfaces have changed over time, and they both have to take on the complexity of being aware and resilient in the face of multiple versions. In the managed add-in model, we are introducing a new architecture that allows the hosts and add-ins to each program against their "view" of the object model and provides host developers a way to put the version-to-version adaptation logic in a separate and isolated assembly.

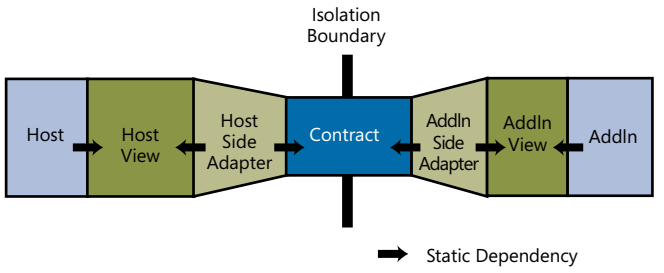
Pipeline architecture

Figure 1 illustrates the architecture we've designed for facilitating versionable communication between hosts and add-ins. The goal is to provide a stable "view" of the object model to both hosts and add-ins with layers abstracting how that view is marshaled across the isolation boundary, how it is versioned, and even how it is consumed on the other side.

Each box in Figure 1 actually represents an assembly that contains many different types. Typically, for every custom object type exchanged between the add-in and the host, there will be at least one corresponding type in each view, adapter, and contract component. The host and the add-in are the entities which add real functionality to the system, while the other components are there to represent the object model, marshal it across the isolation boundary, and adapt it as necessary.

Looking at Figure 1, several very important properties of this

Figure 1: The Host/Add-In Communication Pipeline



architecture begin to pop out. First and foremost is the fact that the host/add-in only depends on its view; the view has no dependencies on other pipeline components. This provides the abstraction barrier that allows the host to be completely agnostic of how the add-in, and the communication mechanism between them, is actually implemented—and vice versa. In fact, if you look at the static dependencies, you will note that the entire center section—from adapter to adapter—can be completely replaced without the host or add-in even being aware of it.

Another important facet of this architecture is that this model is in fact completely symmetric across the isolation boundary, a fact illustrated in Figure 2. This means that from the architectural standpoint there is no difference between how hosts and add-ins communicate with each other and version over time. In fact, in our model the only real difference between a host and an add-in is that the host is the one that happens to activate the other: Once activation has taken place, they become nearly equal peers in the system.

Now that we've described the overall architecture, we can delve into specifics of each pipeline component type and its role in the system.

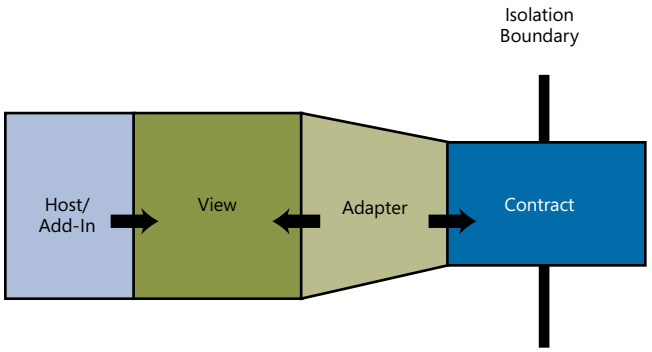
- *The view component.* The "view" component represents, quite literally, the host's and add-in's views of each other and the types exchanged between them. These components contain the abstract base classes and interfaces that each side will program against and are, in essence, each side's SDK for the other. Hosts and add-ins are statically bound to a particular version of these view assemblies and are thus guaranteed to be able to code against this static view regardless of how the other side was implemented. In V1 of an application, these view assemblies can be very similar (in fact they can be the same assembly), but over time, they can drift—in fact, even in some V1 applications, it can be very useful to have a radically different view for each side. It becomes the job of the other components to make sure that even radically different view assemblies can communicate with each other.
- *The contract component.* The "contract" component is the only component whose types actually cross the isolation boundary. Its job is a little funny in that it's not actually a contract between the host and the add-in: It can't be, because the host and the add-in never know which contract component is actually used. Instead, the "contract" component actually represents a contract between the two adapters and facilitates the communication between them across the isolation boundary. Because this is the only assembly that gets loaded on both sides of the boundary, it is important the contract assembly itself never versions: If the adapter on one side expected one version of a contract and the adapter on the other side expected a different version, the contract would be broken and communication would fail.

  Although we say a contract can never version, we are not saying that the host and add-in must always use the same one. Since the contract is in fact a contract between adapters, and not the host/add-in, you can change the contract whenever you want; you simply need to build new adapters that can handle that new contract.

  The contract's special role in our system requires restrictions on the types that are allowed to be contained inside. At a high level, all types defined in the contract assembly must either be interfaces implementing System.AddIn.Contract.IContract or be serializable value types implementing version-tolerant serialization. In addition, the members of these type—most visibly the parameters and return values on their methods—must also be types following the same rules. These restrictions ensure that all reference types that cross the boundary implement at least the base set of functionality in IContract required by the system and that the value types are serializable and can be serialized across versions.
- *The adapter component.* The adapter component's job is to adapt between the view types and the contract types. In reality there are two types of adapters in each adapter component: view-to-contract and contract-to-view. In almost all cases both host and add-in side adapters contain both types since, in most object models, objects are passed both from the host to the add-in and from the add-in to the host. The adapter does its job by implementing the destination type in terms of its source type.

  View-to-contact adapters must adapt from a view type and into a contract type and thus they will take the view type into their constructor and implement the contract interface by calling into that view. If the input or return values to the methods on the contract interfaces are not primitive types that can be directly passed to and returned from the view, then it will be the adapter's job to call upon other adapters to deal with those types. The contract-to-view adapter does the exact opposite job as the view-to-contract adapter and takes in the contract into its constructor and implements the view by calling into that contract. Again, it takes on the responsibility for calling other adapters when the input/return types warrant.

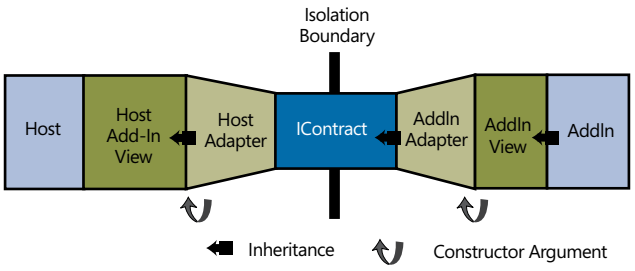
Figure 2: Symmetry of Communication Pipeline



Now that we've defined the individual pieces of the pipeline and their roles, we can put them all together and show how a type gets transferred from one side to another:

- Start with the instance of the type to be passed: defined by the host or the add-in.

Figure 3: Activation Pathway



- Pass the instance to the adapter typed as its view.
- Construct a view-to-contract adapter with the instance.
- Pass the adapter across the isolation boundary typed as the contract.
- Construct a contract-to-view adapter using the contract.
- Pass the adapter back to the other side typed as the view.

This may seem like a lot of overhead, but it actually only involves two object instantiations. In the context of a call across an isolation boundary (AppDomain or Process), the performance overhead is generally much less than the margin of error of the measurements.

In the steady state, all of the above steps are handled by the chain of adapters already active in the connections between the host and add-in. The system itself only steps in to activate the add-in, pass it back to the host, and thus form the first connection. This activation step is just an example of an object being passed from the add-in side to the host (Figure 3). Notice this diagram actually has slightly different names for some of the boxes because it is referring to specific types within each pipeline component. We only use these names for the types used in the activation pathway because these are the only types in the assemblies that the system infrastructure actually deals with directly.

Interesting versioning scenarios

Typically, the host's view changes over time as new versions of the application are released. It may request additional functionality or information from its add-ins, or the data types that it passes may have more or fewer fields and functions. Previously, because of the tight coupling between the host and add-ins (and their data types), this generally meant that add-ins built on one version of an application couldn't be used on later versions and vice versa. Our new architecture was designed from the ground up to provide the right abstraction layers to allow hosts and add-ins talking to radically different object models to nevertheless communicate and connect with each other.

New host / old add-In. The first versioning scenario that comes into everyone's mind is a new host, with a revised object model, trying to run add-ins built against previous versions of the host (Figure 4). With our new model, if the host view changes from one version to the next, the developer just has to create a second AddInSideAdapter, which inherits from the new Contract and converts them to the AddInView version.

The benefit of this approach is that the old add-ins will just keep working, even with the new changes. The application itself doesn't have to keep track of different versions of the add-ins and only deals with a single host view; the different pipeline components connect to either the old or new add-in view. In these cases, the host is tightly coupled to its view, the add-in is tightly coupled to a different view, but versioning is still possible because those views are not tightly coupled to each other.

Old host / new add-in. It will be possible to write an AddInSideAdapter that converts a newer AddInView to the older Contracts. These transformations will be very similar to the ones required to get older add-ins running on newer hosts. (See Figure 5.)

Other pipeline scenarios. In addition to these backwards/forwards compatibility scenarios, other interesting scenarios are enabled by this architecture. One such scenario would be for the pipeline developer to build two separate pipelines for the same view types and optimize them for different isolation boundaries: for example, to build one pipeline that is very fast and used for AppDomain isolated add-ins and use a different one for out-of-process that handles the threading issues.

Or you could go in the opposite direction: Rather than building a pipeline to connect one version of a host to add-ins built against a different version, build one that connects add-ins built for one host to a completely different host. This scenario highlights an important property of this architecture: The developer of the pipeline can be independent from both the host and the add-in, allowing third parties to fill in gaps. If the host developer decides to stop supporting compatibility with the older add-ins, they simply stop building adapters for them, but a customer of the new host (or a hired contractor) could still build and install that adapter, extending the life of the original add-in.

System.AddIn Support for Managed Add-In Hosting

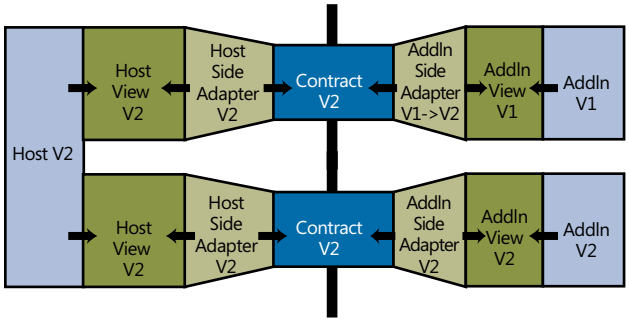
Defining an object model and making it easy for add-in development

When thinking about exposing extensibility points in their applications, the first concerns of many developers are about defining the object model they wish to expose and making sure that the experience for add-in developers is as simple as possible.

We very much believe that the implementation of the host application should be separate from the object model it exposes to its add-ins: This belief led naturally to a design where the exposed object model lives in an assembly separated out from the host (we call it the "view" assembly) and contains a set of classes—typically abstract base classes and interfaces—that represent the host's and add-in's views of each other and of the objects they exchange between them.

This definition of a view assembly with abstract base classes and interfaces leads to a programming model for add-ins that is easy and has low overhead: simply add a reference to the view assembly and then inherit-from/implement the abstract base class or interface the host defined for them. The base class or interface the add-in host specified that the add-ins inherit from is called the "AddInBase." In the simplest case, add-in developers just compile their assemblies and drop them in

Figure 4: Backward Compatibility: V1 Add-In on V2 Host



a directory the host is looking at. Usually the add-in developer will also apply a custom attribute to the add-in to identify it to the host, giving the host information about the add-in before it decides to activate it.

Discovery and activation

Once the application developer has defined the object model, the next steps are to figure out how to discover the available add-ins and activate the ones it wants. Applications will typically perform these actions at startup or in response to a user request, so the performance of these operations is important. Another common requirement for discovery is to be able to query the system for the available add-ins and get information about each add-in before deciding which add-ins to activate.

System.AddIn exposes this functionality through two main classes: AddInStore and AddInToken. AddInStore contains a set of static methods used to find the available add-ins given the type of add-in requested and a set of locations to look for the add-ins. These methods return a collection of AddInTokens which represent available add-ins and provide information about those add-ins. Once you have decided which AddInTokens to activate, you simply call the Activate method on the AddInToken and receive an add-in of the type requested.

In its simplest form, discovery and activation can be performed with just a few lines (in truth, one line of code is missing from this sample):

```
IList<AddInToken> tokens =
    AddInStore.FindAddIns(typeof(AddInType),
    addinPath);
foreach (AddInToken token in tokens)
{
    token.Activate<AddInType>(
    AddInSecurityLevel.Internet);
}
```

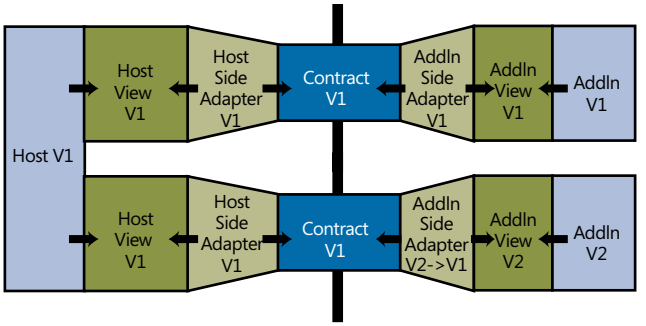
In System.AddIn, we've made sure that applications can include the discovery and activation of add-ins on their startup paths and still have a good user experience. The FindAddIns method is generally safe to call from startup as it doesn't actually load any assemblies (or even read them from disk); instead, it looks in a cache file for the specified directory. This cache file includes information on all the available add-ins in a specific directory as well as the AddInBase of each add-in. Thus, FindAddIns can quickly list out all the available add-ins of a given type in each provided directory by looking only at the single cache file rather than looking at all the assemblies in each directory and enumerating through the types in each assembly looking for add-ins.

Of course, now that FindAddIns provides the guarantee that it only ever looks at a cache file, the question is: When does this cache file get generated? This is where the missing line comes in:

```
AddInStore.Update(addinPath);
```

AddInStore.Update will first determine if the cache of the provided location is up-to-date. If the cache is current, then the method will return very quickly; if not, then it will need to rebuild the cache. We also provide a tool as part of the framework (AddInUtil.exe) that makes it easier to update the cache as part of an installation program. By separating out FindAddIns and Update, and providing a command line tool that performs the same action, we give host applications a lot of flexibility.

Figure 5: Forward Compatibility: V2 Add-in on V1 Host



If startup performance is critical, the application can decide to only call FindAddIns on startup and either require that add-in developers use AddInUtil as part of their installation or give users the option to initiate an update of the cache and refresh of available add-ins. On the other hand, the application developer can decide to take a performance hit that only occurs when new add-ins are installed, and call AddInStore. Update on startup.

Isolation/sandboxing

Of course, discovering the add-ins and deciding to activate them is only part of the story: You also need to ensure that the add-ins are activated in the desired "environment." Several things make up this "environment":

Isolation Level:

- In-AppDomain
- Cross-AppDomain
- Cross-Process

Pooling:

- In domain with other, host-specified add-ins
- In its own domain
- Cross-process in a domain with other, host-specified add-ins
- Cross-process in its own domain with other domains in the external process hosting other add-ins
- Cross-process in its own domain, without any other add-ins running in that process

Security level:

- One of the system defaults: Internet, Intranet, FullTrust
- Custom PermissionSet specified by the host.

By taking a look at the overloads available on AddInToken.Activate, you can see how easy it is for hosts to specify the desired environment. At the simplest case, the host simply needs to pass in an enum value specifying Internet, Intranet, or FullTrust, and we'll create an AppDomain with that trust level, activate the add-in in it, and then pass it back. The level of difficulty depends on how much control the host wants: On the high end, it can create an AppDomain on its own, fine-tune it to its needs, and then pass that to us for activation.

Reliable Hosting Techniques

Reliable hosting

One issue of prime importance to many developers of extensible applications is the ability to maintain the reliability of the host in the face of buggy add-ins. In some cases, this is important because the host

application itself is mission-critical and can't accept the prospect of down-time caused by an add-in; in other cases, the host developer just can't afford the cost of supporting customers coming to them for help due to bugs in third-party extensions. Some hosts need simply to be able to identify and disable the add-in that caused the crash while others need to make sure the host will keep running, regardless of what their add-ins do. Each host's reliability requirement must be assessed.

There are generally three categories of concerns for host developers when it comes to reliability: corruption of machine state, unhandled exceptions (crashes), and machine resource exhaustion. Each category poses its own unique problems and requires that different actions be taken.

Machine state corruption

In some ways, corruption of machine state can be the easiest problem to address because the system itself has the most support for preventing it. Generally, all the host needs to do is grant its add-ins as limited a security PermissionSet as possible and the .NET Frameworks Code-Access-Security system will ensure that the add-in stays within those bounds. The APIs in System.AddIn make this even easier either by letting hosts use one of the system defined security levels (Internet, Intranet, or FullTrust) or allowing them to define their own custom PermissionSet. The more system resources you want to protect from your add-ins, the fewer permissions those add-in have when they execute.

Unhandled exceptions

Unhandled exceptions start to get more interesting as there are really two types of unhandled exceptions that the host has to be concerned about. The first type includes the exceptions that get thrown to the host during a call from the host into the add-in and on the same thread. To deal with these exceptions, the host needs to add a try/catch block around all of its calls into the add-in and react appropriately when an exception is thrown. The other exceptions are much harder to deal with as they are actually impossible for a host to catch: These are exceptions that occur on threads originating from the add-in or on threads where the host is not lower down on the stack.

Starting with the 2.0 version of the Common Language Runtime (CLR), these unhandled exceptions will always be fatal to the process; thus, if care is not taken, it becomes very easy for a buggy add-in to take the host down with it. If the host's reliability requirement simply requires disabling problem add-ins (often a perfectly acceptable solution for client applications), the host can take advantage of something on the AppDomain class called the UnhandledException event. This event fires when a thread originating in the target AppDomain throws an unhandled exception that is about to take the process down. If the host simply wants to identify the misbehaving add-ins and disable them, it can subscribe to these events on the AppDomains its add-ins run in; when the event fires, it can record which add-in was running in that AppDomain. The host cannot prevent the process from being taken down, but it will be able to add the add-in to a disabled list and then restart the host.

If the host needs to ensure that a crashing add-in cannot take down the host, then it will need to take further measures and isolate its add-ins in a different process. There is an associated performance hit, but isolation ensures that only the add-in process gets taken down in the case of a failure.

Machine resource exhaustions

The final category of problems the host needs to be aware of is the exhaustion of system resources by the add-in. This can take the form of using excessive amounts memory or even CPU cycles. Within a process, there is no way to limit the resources of add-ins running in different AppDomains, but as soon as you move them into a different process, you can use the operating system to throttle those add-ins. Once you activate the add-in out-of-process, a few additional lines of code are required to achieve this throttling:

```
AddInProcess addInProcess = new AddInProcess();
AddInType addIn =
  token.Activate<AddIntype>(addInProcess,
  AddInSecurityLevel.FullTrust);
Process tmpAddInProcess =
   Process.GetProcessById(addInProcess.ProcessId);
//Lower the priority of the add-in process below that
    of the host
tmpAddInProcess.PriorityClass =
   ProcessPriorityClass.BelowNormal;
//Limit the add-in process to 50mb of physical memory
tmpAddInProcess.MaxWorkingSet = (IntPtr)50000000;
```

Hosts can go even further and use the operating system to monitor everything from the total CPU time consumed by the process to the number of disk I/O requests it is making per second, and then take appropriate action.

Wrapping Up

With the addition of the System.AddIn assemblies and the architecture they support, the Microsoft.NET Framework 3.5 delivers a complete managed add-in model, first promised back in PDC 2005. You can expect to see a few additions to our feature set in future betas and even more in future versions, but all you need to build fully extensible, versionable, and reliable hosts is available now and ready for you to use. To learn more about the topics covered in this article or for general information about this add-in model, please visit the add-in team blog on MSDN (see Resources).

Resources

CLR Add-In Team Blog
http://blogs.msdn.com/clraddins/

"CLR InsideOut: .Net Application Extensibility," Jack Gudenkauf and Jesse Kaplan, MSDN Magazine, February and March, 2007
Part 1, http://msdn.microsoft.com/msdnmag/issues/07/02/CLRInsideOut/
Part 2, http://msdn.microsoft.com/msdnmag/issues/07/03/CLRInsideOut/

Visual Studio Code Name "Orcas"
http://msdn.microsoft.com/vstudio/future

About the Author

Jesse Kaplan is a program manager for application extensibility and runtime versioning on the Common Language Runtime team. His past areas of responsibility include compatibility, managed-native (COM) interoperability, and metadata.

Microsoft®

THE ARCHITECTURE JOURNAL™
Input for Better Outcomes