Architecture Strategies for Catching the Long Tail

Frederick Chong and Gianpaolo Carraro
Microsoft Corporation

April 2006

Applies to:
   Application Architecture
   Software-as-a-Service (SaaS)

Summary: This provides an overview of the software-as-a-service (SaaS) model for software delivery, provides a high-level description of the architecture of a SaaS application, and discusses the challenges and benefits of developing and offering SaaS. (26 printed pages)

For further reference, an ARCast is available:

ARCast – Software As A Service

Contents

Introduction

Software as a service. The words are on everyone's lips. The pages of software industry publications are full

of articles about software as a service (SaaS)—articles that use words like "revolution" and "horizon" (as in, "on the…"). Everyone knows (or thinks they know) what it is, roughly, and everyone knows it's going to be big. Yet few people would say they can really define it, and even fewer know how to build it.

So, if SaaS holds such promise for the future of application delivery, why isn't there more guidance available to help people actually achieve it?

We believe that SaaS *is* going to have a major impact on the software industry, because software as a service will change the way people build, sell, buy, and use software. For this to happen, though, software vendors need resources and information about developing SaaS applications effectively.

This is the first in a series of papers from Microsoft dedicated to demystifying SaaS and providing practical, real-world guidance for architecting SaaS applications. This paper serves as an overview of SaaS, its challenges, and its benefits for those who are interested in offering SaaS. Future papers will explore many of these topics in detail.

This paper begins by asking just *what* software as a service is, exactly, and it explains the conceptual shifts that prospective SaaS vendors must experience in order to understand how it differs from traditional, on-premise software. Next, we'll look at the SaaS business model, to see how software as a service can be monetized in the real world.

Because this is an architectural paper, the largest section addresses the architecture of a SaaS application. We present a four-level maturity model that explains and puts into perspective some key attributes of SaaS: configurability, multi-tenant efficiency, and scalability. We'll examine the components of a high level SaaS architecture, and then take a closer look at a typical challenge the SaaS architect faces—that of providing a mechanism for extending the data model of a multi-tenant application.

Lastly, we'll take a brief look at some of the operational issues involved in supporting a SaaS application after deployment.

What Is Software as a Service?

Even today, the exact definition of software as a service (SaaS) is open to debate, and asking five people would probably result in five different definitions. Still, most experts would probably agree on a few fundamental principles that distinguish SaaS from traditional packaged software on the one hand, and simple websites on the other. Expressed most simply, software as a service can be characterized as follows:

"Software deployed as a hosted service and accessed over the Internet."

Take a moment to consider the implications of this definition. It doesn't prescribe any specific application architecture; it doesn't say anything about specific technologies or protocols; it doesn't draw a distinction between business-oriented and consumer-oriented services, or require specific business models. According to this definition, the key distinguishing features of software as a service are where the application code resides, and how they are deployed and accessed.

(Is this definition a little simplistic? In a word… yes. Later, we'll focus on some of the attributes that define and distinguish a well-designed, mature SaaS application.)

By this definition, SaaS includes a number of services and applications that you may not expect to find in this category. For example, consider Web-based e-mail services, such as Microsoft Hotmail. Although Hotmail might not be the first example that comes to mind when you think about SaaS, it meets all of the basic criteria: a vendor hosts all of the program logic and data, and provides end users with access to this

data over the public Internet, through a Web-based user interface.

Moving from the general to specific, we can identify two major categories of software as a service:

- **Line-of-business services**, offered to enterprises and organizations of all sizes. Line-of-business services are often large, customizable business solutions aimed at facilitating business processes such as finances, supply-chain management, and customer relations. These services are typically sold to customers on a subscription-basis.
- **Consumer-oriented services**, offered to the general public. Consumer-oriented services are sometimes sold on a subscription-basis, but are often provided to consumers at no cost, and are supported by advertising.

This paper focuses on the architecture and business issues involved in developing line-of-business applications, and the concepts and examples herein are presented in that context. However, issues such as multi-tenant customization and extensibility, data scaling, and isolation issues also occur (and in fact tend to be easier to resolve) in the consumer space, so developers of consumer-oriented SaaS offerings may benefit from reading it as well.

Thinking About Software as a Service

Moving from offering on-premise software to offering software as a service requires software vendors to shift their thinking in three interrelated areas: in the business model, in the application architecture, and in the operational structure (see Figure 1).
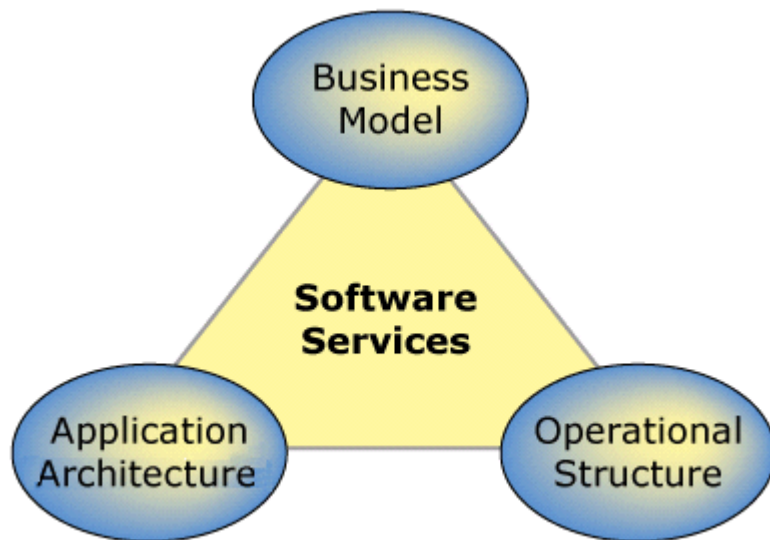

Figure 1. Areas in which software vendors need to shift their thinking

In the following three sections, we'll take a closer look at each of these shifts, focusing primarily on the application architecture aspect of SaaS.

Changing the Business Model

Changing the business model could involve one or more of the following:

- Shifting the "ownership" of the software from the customer to an external provider.
- Reallocating responsibility for the technology infrastructure and management—that is, hardware and

professional services—from the customer to the provider.
- Reducing the cost of providing software services, through specialization and economy of scale.
- Targeting the "long tail" of smaller businesses, by reducing the minimum cost at which software can be sold.

Realizing the benefits of SaaS requires shifts in thinking on the part of both the provider and the customer, and it's up to the provider to help the customer make this shift.

Who "Owns" the Software?

Most software continues to be sold in the same way it has been sold for decades. The customer buys a license to use the software, and installs it on hardware that belongs to the customer or that is otherwise under the customer's control, with the vendor providing support as directed by the terms of the license or a support agreement. In an honest, above-board software transaction, the notion of a "license" can seem like something of a technicality: legally, the customer is only purchasing the right to use a copy of the software, but for practical purposes, it's as though the customer "owns" the software and may use it as often and for as long as it wishes.

With the software-as-a-product model providing the context for the software market, the idea of software as a service can feel somewhat alien: instead of "owning" important software outright, customers are told, they can pay for a subscription to software running on someone else's servers, software that goes away if they stop subscribing. It's therefore especially important that the prospective customer understand how SaaS provides a direct and quantifiable economic benefit over the traditional model.

Transferring IT Responsibilities

In a typical organization, the information technology (IT) budget is spent in three broad areas:

- **Software**—The actual programs and data that the organization uses for computing and information processing.
- **Hardware**—The desktop computers, servers, networking components, and mobile devices that provide users with access to the software.
- **Professional services**—The people and institutions that ensure the continued operation and availability of the system, including technical support staff, consultants, and vendor representatives.

Of these three, it is the software that is most directly involved in information management, which is the ultimate goal of any IT organization. Hardware and professional services, though vital and important components of the IT environment, are properly considered means to an end, in that they make it possible for the software to produce the desired end result of effective information management. (To put it another way, any organization would gladly add software functionality without extra hardware if it could do so effectively, but no organization would simply add hardware without an anticipated need to add software as well.)

In an IT environment based around on-premise software, the majority of the budget is typically spent on hardware and professional services, leaving a minority of the budget available for software (see Figure 2).
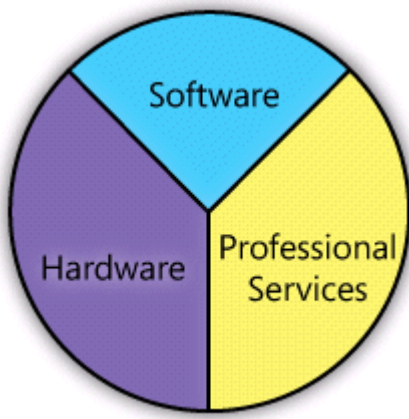
Figure 2. Typical buget for an on-premise software environment

In this model, the software budget is spent primarily on licensed copies of "shrink-wrapped" business software and customized line-of-business software. The hardware budget goes toward desktop and mobile computers for end users, servers to host data and applications, and components to network them together. The professional services budget pays for a support staff to deploy and support software and hardware, as well as consultants and development resources to help design and build custom systems.

> **Note** The proportions shown in these diagrams are for illustrative purposes only; they are not intended to advocate any specific allocation of resources, and your allocation may differ significantly.

In an organization relying chiefly on SaaS, the IT budget allocation looks much different (see Figure 3).
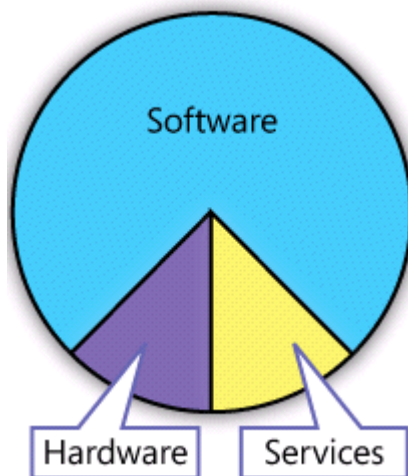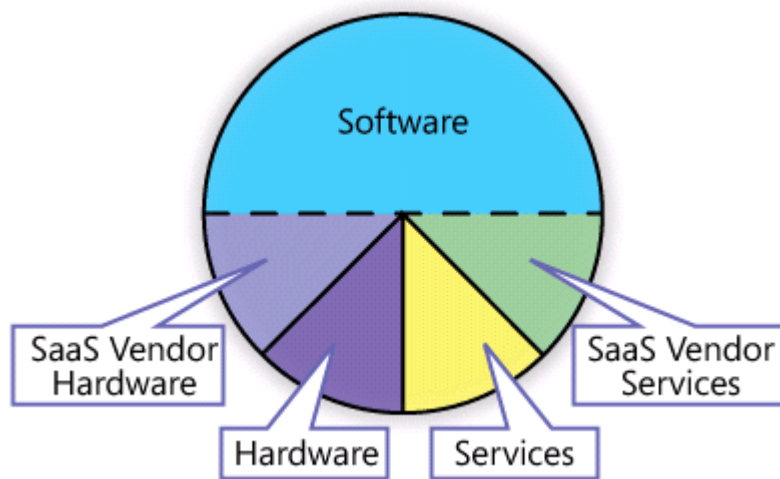

Figure 3. Typical budget for an SaaS environment

In this model, the SaaS vendor hosts critical applications and associated data on central servers at the vendor's location, and it supports the hardware and software with a dedicated support staff. This relieves the customer organization from the responsibility for supporting the hosted software, and for purchasing and maintaining server hardware for it. Moreover, applications delivered over the Web or through smart clients place significantly less demand on a desktop computer than traditional locally-installed applications, which enables the customer to extend the desktop technology lifecycle significantly. The end result is that a much larger percentage of the IT budget is available to spend on software, typically in the form of

subscription fees to SaaS providers.

Leveraging Economy of Scale

But isn't this result just an illusion? After all, a percentage of the subscription fees paid to SaaS vendors for "software" has to pay for hardware and professional services for the vendor. The answer lies in the economy of scale. A SaaS vendor with x number of customers subscribing to a single, centrally-hosted software service enables the vendor to serve all of its customers in a consolidated environment. For example, a line-of-business SaaS application installed in a load-balanced farm of five servers may be able to support 50 medium-sized customers, meaning that each customer would only be responsible for a tenth of the cost of a server. A similar application installed locally might require each customer to dedicate an entire server to the application—perhaps more than one, if load balancing and high availability are concerns. This represents a substantial potential savings over the traditional model, and for SaaS applications that are built to scale well, the operating cost for each customer will continue to drop as more customers are added. As this is happening, the provider will develop multi-tenancy as a core competency, leading to higher-quality offerings at a lower cost. Therefore, even accounting for the hardware and professional services costs incurred by SaaS vendors, customers can still obtain significantly greater pure software functionality for the same IT budget (see Figure 4).

Figure 4. Typical budget for an SaaS environment (accounting for hardware and professional services costs)

Selling to the Long Tail

With his article "The Long Tail," in the October 2004 issue of *Wired* (http://www.wired.com/wired/archive/12.10/tail.html), writer Chris Anderson popularized the idea of the "long tail" in explaining why online retailers such as Amazon.com are uniquely positioned to fill a huge demand that traditional retailers cannot serve cost-effectively (see Figure 5).
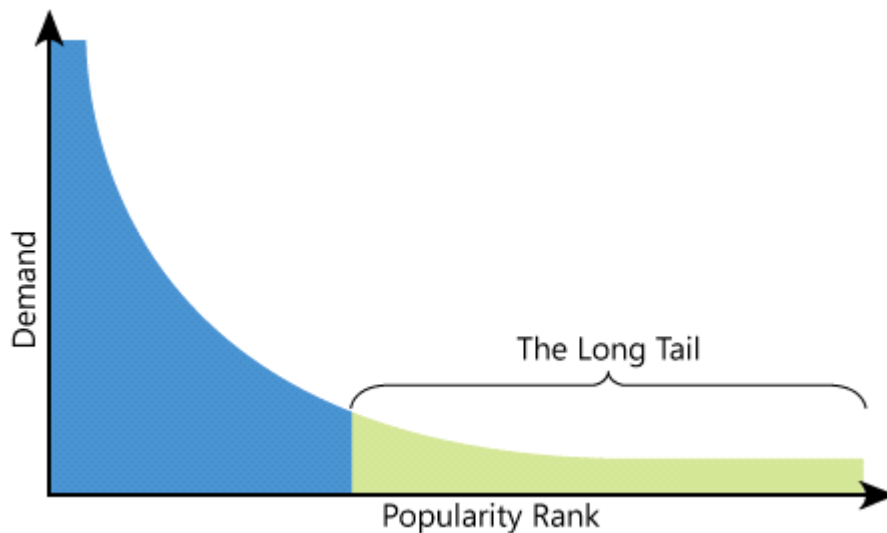
Figure 5. The "long tail"

Demand for categories of merchandise such as books or compact discs tends to follow what is known as a "power law distribution." In this type of scenario, thousands of books, CDs, and DVDs are published every year, but only a few dozen titles ever rise to the level of bestseller. The rest languish in the so-called long tail: the huge number of smaller releases with specialty appeal that can never hope to sell more than a few thousand copies, perhaps not even that many.

Traditional "brick-and-mortar" retailers concentrate on selling the most popular items, because they can't possibly stock copies of each of the millions of books, CDs, and DVDs in print. Online retailers, however, don't have to worry about limited shelf space; shipping items to customers directly from large warehouses around the world, they can advertise and sell the millionth most popular title as easily as the most popular one. Access to this long tail of low-volume sales translates into a huge amount of revenue.

A large brick-and-mortar bookstore might carry about 130,000 different titles on its shelves. Yet, according to Anderson, the majority of Amazon.com's book sales come from *outside* its top 130,000 titles—in other words, most of the books that Amazon.com sells are titles that wouldn't even be carried by a traditional walk-in bookstore.

Vendors of complex line-of-business (LOB) software solutions face a similar market curve (see Figure 6).
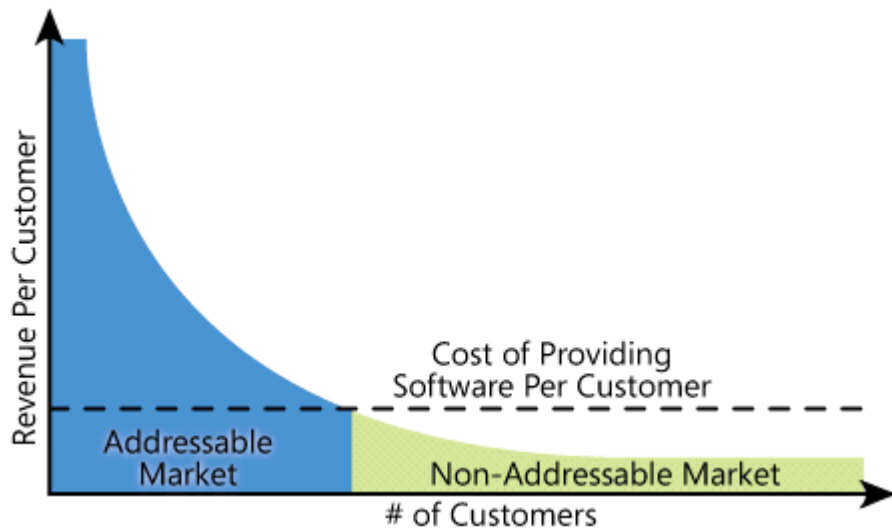
Figure 6. Market curve for LOB software vendors

In contrast to simpler, shrink-wrapped software packages, line-of-business software tends to be custom-tailored to meet individual customers' needs—potentially including on-site installation and service visits from vendor service teams—and often requires dedicated server hardware, and support staff to manage it. The cost of providing this kind of dedicated attention contributes to the minimum price at which the vendor can afford to sell the software. Such software therefore tends to be marketed toward larger businesses that can afford to pay for this level of attention. But for every large enterprise that purchases a line-of-business solution, there are dozens of smaller and medium-sized businesses that could benefit from such a solution, but that cannot afford the expense.
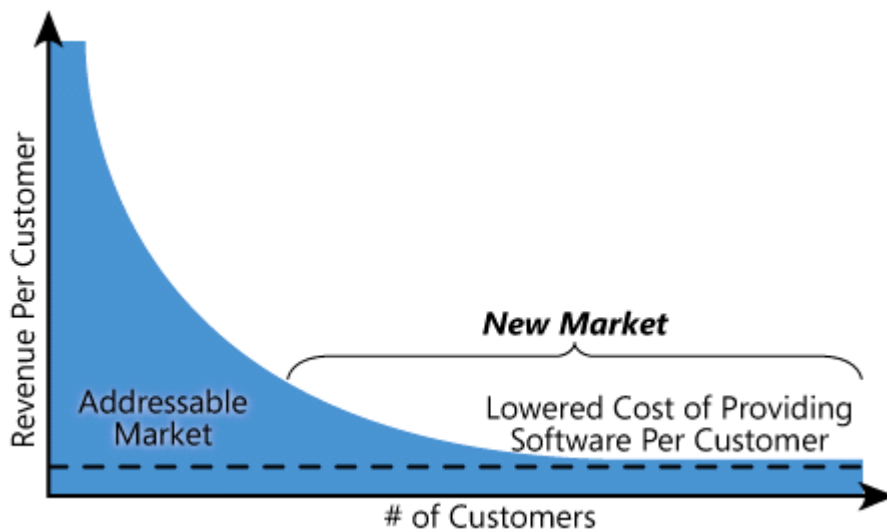
Figure 7. New market opened by lower cost of SaaS

By eliminating much of the upkeep, and using the economics of scale to combine and centralize customers' hardware and services requirements, SaaS vendors can offer solutions at a much lower cost than traditional vendors, not only in monetary terms, but also by greatly reducing the need for customers to add complexity to their IT infrastructure. This gives SaaS exclusive access to an entirely new range of potential customers that have always been inaccessible to traditional solution vendors, because it has never before been cost-effective to serve them (see Figure 7).

Effectively targeting these smaller customers requires another shift in thinking for vendors who are accustomed to a sales process that depends on personal contacts and vendor-customer relationships; most vendors won't be able to provide personal service to a much larger customer base at price points that such a base will support. Selling SaaS is like selling mobile phone ringtones, or downloadable music: it should be possible for a customer to visit your website, subscribe to your service, pay with a credit card, customize the service, and begin using it, all without human intervention on the part of the vendor. This doesn't mean that you have to eliminate the more personal approach for larger customers with more extensive needs. But designing the sales, marketing, provisioning, and customization processes from the ground up to work automatically makes it possible to offer an automated approach as a choice—and has the happy side effect of simplifying the work that your own support personnel must perform in order to accomplish the same tasks on behalf of a customer.

Application Architecture

Our working definition of software as a service is: "Software deployed as a hosted service and accessed over the Internet." Depending on how one defines words such as *software* and *access*, this definition can encompass a lot of things… perhaps too many. To an application architect, certainly, it doesn't really shed any light on what exactly makes a SaaS application work, the thing that makes the difference between a successful SaaS application and an unsuccessful one. A line-of-business application with a decade-old code base mated to a jury-rigged HTML front end may fit the broad definition of software as a service, but most such applications run into problems when they are unable to scale well or cost-effectively. To define what might be called a mature SaaS application, therefore, we must introduce some additional criteria.

The Three Attributes of a Single-Instance Multi-Tenant Architecture

From an application architect's point of view, there are three key differentiators that separate a well-designed SaaS application from a poorly designed one. A well-designed SaaS application is *scalable*, *multi-tenant-efficient*, and *configurable*.

Scaling the application means maximizing concurrency, and using application resources more efficiently—for example, optimizing locking duration, statelessness, sharing pooled resources such as threads and network connections, caching reference data, and partitioning large databases.

Multi-tenancy may be the most significant paradigm shift that an architect accustomed to designing isolated, single-tenant applications has to make. For example, when a user at one company accesses customer information by using a CRM application service, the application instance that the user connects to may be accommodating users from dozens, or even hundreds, of other companies—all completely unbeknownst to any of the users. This requires an architecture that maximizes the sharing of resources across tenants, but that is still able to differentiate data belonging to different customers.

Of course, if a single application instance on a single server has to accommodate users from several different companies at once, you can't simply write custom code to customize the end-user experience —anything you do to customize the application for one customer will change the application for other customers as well. Instead of *customizing* the application in the traditional sense, then, each customer uses metadata to *configure* the way the application appears and behaves for its users. The challenge for the SaaS architect is to ensure that the task of configuring applications is simple and easy for the customers, without incurring extra development or operation costs for each configuration.

The Software as a Service Maturity Model

We've enhanced our working definition of SaaS by identifying the important attributes of a mature SaaS application. But maturity isn't an all-or-nothing proposition. An application can possess just one or two of these attributes and still meet all necessary business requirements, in which case the application architects may actively choose not to fulfill the other attributes, if doing so would not be cost-effective.

Broadly speaking, SaaS application maturity can be expressed using a model with four distinct levels. Each level is distinguished from the previous one by the addition of one of the three attributes listed above.
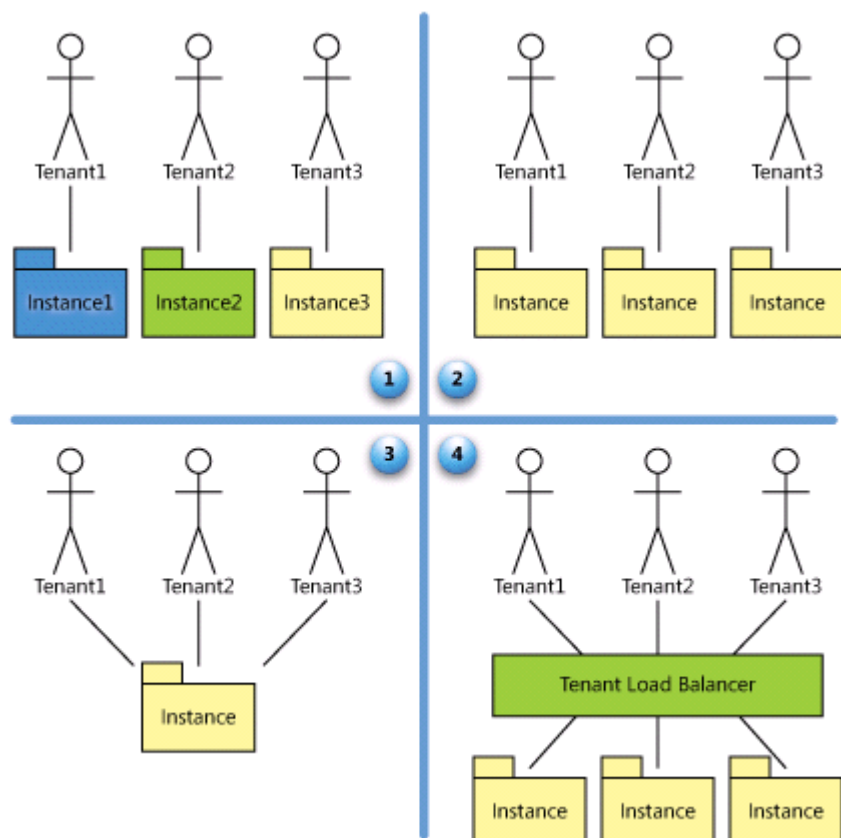
Figure 8. Four-level Saas maturity model

Level I: Ad Hoc/Custom

The first level of maturity is similar to the traditional application service provider (ASP) model of software delivery, dating back to the 1990s. At this level, each customer has its own customized version of the hosted application, and runs its own instance of the application on the host's servers. Architecturally, software at this maturity level is very similar to traditionally-sold line-of-business software, in that different clients within an organization connect to a single instance running on the server, but that instance is wholly independent of any other instances or processes that the host is running on behalf of its other customers.

Typically, traditional client-server applications can be moved to a SaaS model at the first level of maturity, with relatively little development effort, and without re-architecting the entire system from the ground up. Although this level offers few of the benefits of a fully mature SaaS solution, it does allow vendors to reduce costs by consolidating server hardware and administration.

Level II: Configurable

At the second level of maturity, the vendor hosts a separate instance of the application for each customer (or *tenant*). Whereas in the first level each instance is individually customized for the tenant, at this level, all instances use the same code implementation, and the vendor meets customers' needs by providing detailed configuration options that allow the customer to change how the application looks and behaves to its users. Despite being identical to one another at the code level, each instance remains wholly isolated from all the others.

Moving to a single code base for all of a vendor's customers greatly reduces a SaaS application's service requirements, because any changes made to the code base can be easily provided to all of the vendor's customers at once, thereby eliminating the need to upgrade or slipstream individual customized instances. However, repositioning a traditional application as SaaS at the second maturity level can require significantly more re-architecting than at the first level, if the application has been designed for individual customization rather than configuration metadata.

Similarly to the first maturity level, the second level requires that the vendor provide sufficient hardware and storage to support a potentially large number of application instances running concurrently.

Level III: Configurable, Multi-Tenant-Efficient

At the third level of maturity, the vendor runs a *single* instance that serves every customer, with configurable metadata providing a unique user experience and feature set for each one. Authorization and security policies ensure that each customer's data is kept separate from that of other customers; and, from the end user's perspective, there is no indication that the application instance is being shared among multiple tenants.

This approach eliminates the need to provide server space for as many instances as the vendor has customers, allowing for much more efficient use of computing resources than the second level, which translates directly to lower costs. A significant disadvantage of this approach is that the scalability of the application is limited. Unless partitioning is used to manage database performance, the application can be scaled only by moving it to a more powerful server (scaling up), until diminishing returns make it impossible to add more power cost-effectively.

Level IV: Scalable, Configurable, Multi-Tenant-Efficient

At the fourth and final level of maturity, the vendor hosts multiple customers on a load-balanced farm of identical instances, with each customer's data kept separate, and with configurable metadata providing a unique user experience and feature set for each customer. A SaaS system is scalable to an arbitrarily large number of customers, because the number of servers and instances on the back end can be increased or decreased as necessary to match demand, without requiring additional re-architecting of the application, and changes or fixes can be rolled out to thousands of tenants as easily as a single tenant.

Choosing a Maturity Level

What maturity level should you target for your application? One might expect the fourth level to be the ultimate goal for any SaaS application, but this isn't always the case. It may be more helpful to think of SaaS maturity as a continuum between *isolated* data and code on one end, and *shared* data and code on the other (see Figure 9).

Figure 9. SaaS maturity as a continuum

Where your application should fall along this continuum depends on your business, architectural, and operational needs, and on customer considerations. As you'll be able to see even from this simple explanation, all of these considerations are interrelated to some degree.

- **Business model**—Does an isolated approach make financial sense? Forsaking the economic and management benefits of a shared approach means offering your application to the consumer at a higher cost; however, under some circumstances, it may be worth it to meet other needs. In addition, customers may have strong legal or cultural resistance to an architectural model in which multiple tenants share access to an application, even if you can demonstrate that it does not place confidential data at risk. Ultimately, of course, you'll need a business model that shows how your application can make money at whichever maturity level you've targeted.
- **Architectural model**—Can your application be made to run in a single logical instance? If you are seeking to move a desktop-based or traditional client-server application to an Internet-based delivery system, it may be fundamentally incompatible with a single-instance, metadata-centric approach, and you may determine that it will never make financial sense to invest the development effort necessary to transform it into a fully mature SaaS application. If you are designing and building a net-native application from the ground up, you will probably have a lot more freedom to take a single-instance approach.
- **Operational model**—Can you guarantee your service level agreements (SLAs) without isolation? Carefully examine the obligations imposed by any existing SLAs that you have with customers, with regard to considerations such as downtime, support options, and disaster recovery, and determine whether these obligations can be met under an application architecture in which multiple unrelated customers share access to a single application instance.

High-Level Architecture

Architecturally, SaaS applications are largely similar to other applications built using service-oriented design principles (see Figure 10).
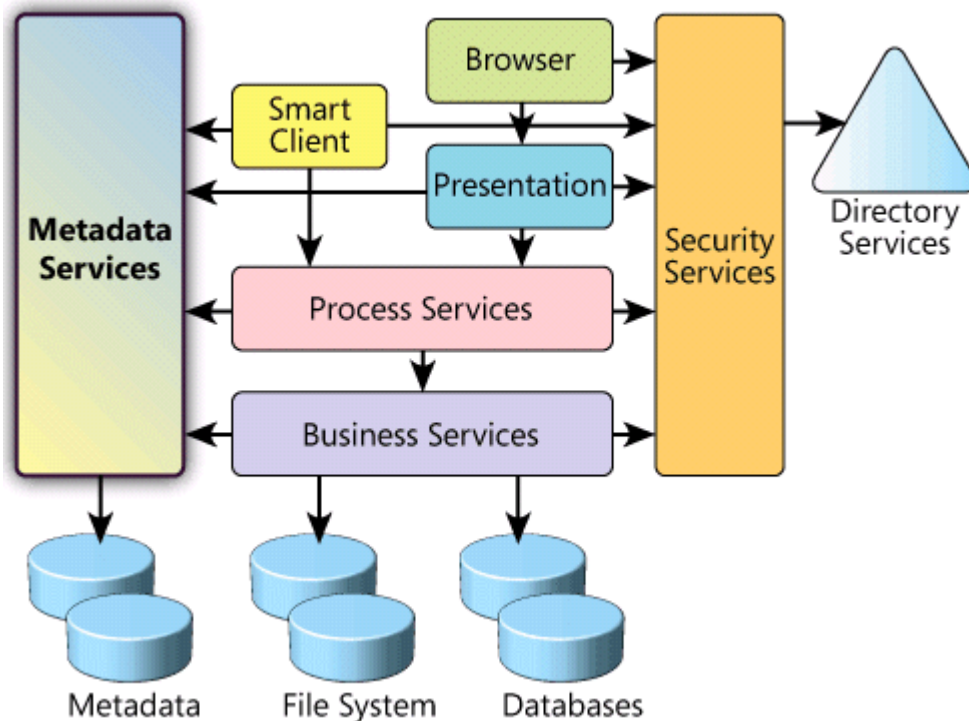
Figure 10. SaaS application architecture

Most of the components depicted in Figure 10 should be familiar to most application architects. The process services expose interfaces that smart clients and/or the Web presentation tier can invoke, and kick off a synchronous workflow or long–running transaction that will invoke other business services, which interact with the respective data stores in order to read and write business data. Security services are responsible for controlling access to end–user and back–end software services.

The most significant difference is the addition of metadata services, which are responsible for managing application configuration for individual tenants. Services and smart clients interact with the metadata services in order to retrieve information that describes configurations and extensions that are specific to each tenant.

Metadata Services

In a mature SaaS application, the metadata service provides customers with the primary means of customizing and configuring the application to meet their needs. Typically, customers can make configuration changes in four broad areas:

- **User interface and branding**—Customers often appreciate the ability to modify the user interface to reflect their corporate branding, and therefore SaaS applications typically offer features that allow customers to change things such as graphics, colors, fonts, and so on.
- **Workflow and business rules**—To be of use to a wide range of potential customers, a business–critical SaaS application has to be able to accommodate differences in workflow. For example, one customer of an invoice tracking application may require each invoice to be approved by a manager; a second customer may require each invoice to be approved by two managers in sequence; a third may require two managers to approve each invoice, but allow them to work in parallel. When appropriate, customers should be able to configure the way in which the application's workflow aligns with their business processes.
- **Extensions to the data model**—For many data-driven SaaS applications, one size definitely doesn't

fit all. Even with relatively simple, task-specific applications, customers may chafe under the restrictions imposed by a static, unchanging set of data fields and tables. An extensible data model gives customers the freedom to make an application work their way, instead of forcing them to work *its* way. Later in this paper, you'll learn a bit more about how a customer-extensible data model is architected.

- **Access control**—Typically, each customer is responsible for creating individual accounts for end users, and for determining which resources and functions each user should be allowed to access. Access rights and restrictions for each user are tracked by using security policies, which should be configurable by each tenant.

To provide customers with flexibility in configuring the software as necessary, these options are organized into hierarchical configuration units known as *scopes*, each of which contains options for making changes in each of the four areas listed above. Every customer has a top-level scope that it can configure as needed, and the customer may establish one or more scopes underneath the top level in an arbitrary hierarchy. A *relationship strategy* determines how and whether child nodes inherit and override configuration settings from parent nodes.

For example, a typical customer that purchases enterprise-wide access to your application may have several business units with distinct needs, all of which must follow certain company-wide standards, but also must be able to configure some aspects of the application individually. Within each business unit as well, there may be organizational groups that have their own special configuration needs. For each of these identified organizational units, the customer can establish a scope that gives the group access to the configuration options that it may set or change.

Unlike traditional vendor-customized line-of-business applications, SaaS applications are much more likely to be configured by customers themselves. Designing the configuration interface is therefore almost as important as designing the interface for end users. Ideally, customers should be able to configure the application through a wizard, or through simple, intuitive screens that present all available options without causing information overload, and that clearly distinguish between options that can and cannot be changed within a given scope.

Security Services

As important as security is in any software context, the nature of SaaS makes security both a paramount concern for customers, and a high priority for application architects. Following some basic guidelines can help ensure that tenants remain in control of their private data.

Authentication

The SaaS provider typically delegates to each tenant the responsibility for creating and maintaining its own user accounts, a process known as *delegated administration*. Delegated administration creates a situation in which the customer is responsible for creating individual user accounts, but the vendor has to authenticate them. To accommodate this delegated-administration model, SaaS designers use two general approaches for handling authentication: a centralized authentication system, or a decentralized authentication system. The approach that you choose will have ramifications for the complexity of your architecture and the way end users experience the application, and you should consider what your business model says about the needs of the application, customers, and end users when making a decision.

In a centralized authentication system, the provider manages a central user account database that serves all of the application's tenants. Each tenant's administrator is granted permission to create, manage, and delete user accounts for that tenant in the user account directory. A user signing on to the application provides his

or her credentials to the application, which authenticates the credentials against the central directory and grants the user access if the credentials are valid (see Figure 11).
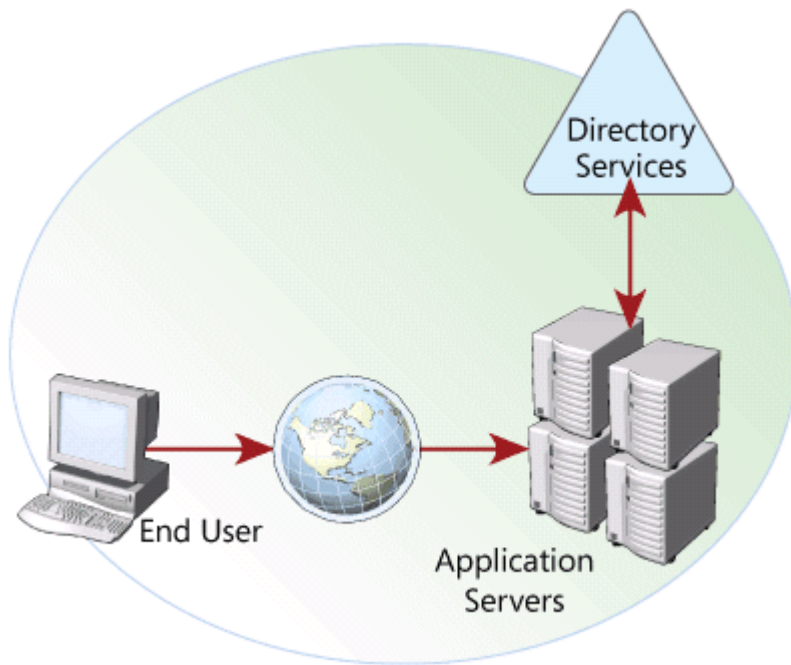
Figure 11. Centralized authentication system

This approach requires a relatively simple authentication infrastructure that is comparatively easy to design and implement, and that does not require any changes to the tenant's own user infrastructure. An important disadvantage to this approach is that a centralized authentication system makes it much more difficult to implement *single sign-on*, in which the application accepts the credentials that the user has already entered to gain access to his or her corporate network. Without single sign-on, users are frequently presented with an inconvenient login prompt when logging in to the application, and they must enter their credentials manually.

In a decentralized authentication system, the tenant deploys a federation service that interfaces with the tenant's own user directory service. When an end user attempts to access the application, the federation service authenticates the user locally and issues a security token, which the SaaS provider's authentication system accepts and allows the user to access the application (see Figure 12).
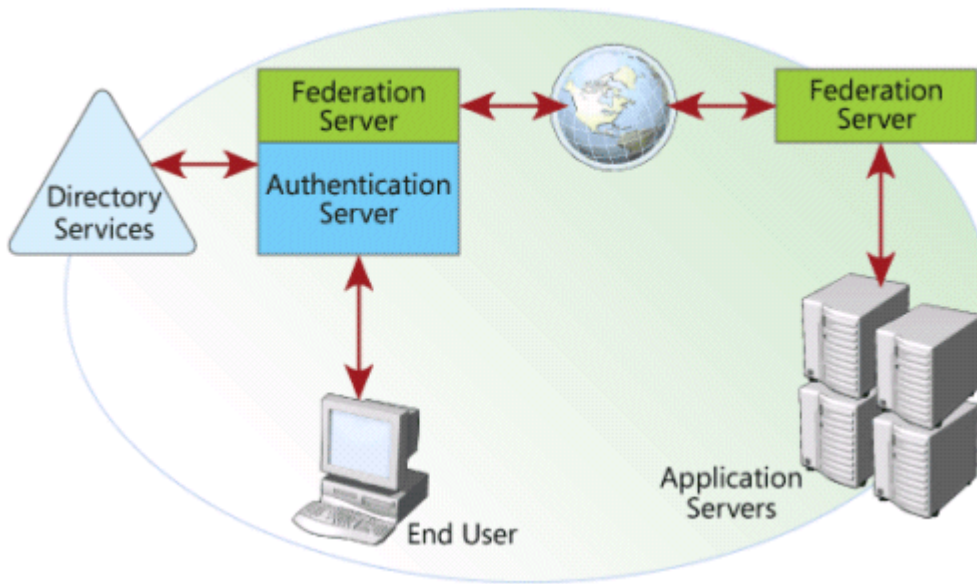
Figure 12. Decentralized authentication system

This is an ideal approach when single sign–on is important, because authentication is handled behind the scenes, and it doesn't require the user to remember and enter a special set of credentials. The decentralized approach is more complex than the centralized approach, however, and a SaaS application with thousands of customers will require individual trust relationships with each of the thousands of tenant federation services.

In many cases, the SaaS provider may want to consider a hybrid approach—using the centralized approach to authenticate and manage users of smaller tenants, and the federated approach for larger enterprises that demand, and will pay for, the single sign–on experience.

Authorization

Typically, access to resources and business functions in a SaaS application is managed by using *roles* that map to specific job functions within an organization. Each role is given one or more *permissions* that enable users assigned to the role to perform actions in accordance with any relevant *business rules* (see Figure 13).
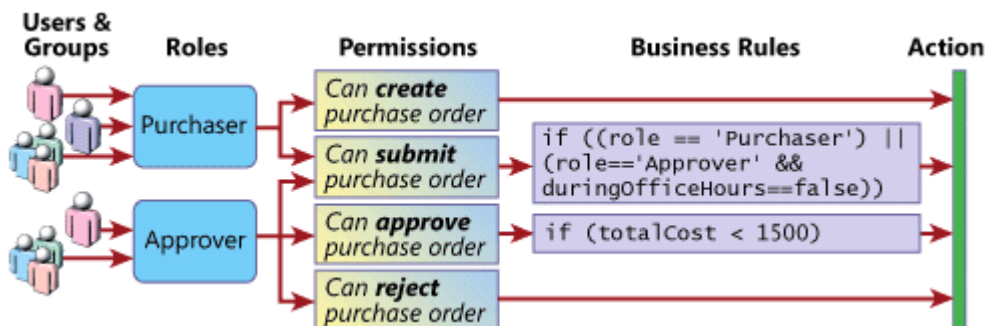
Figure 13. Access control

Roles are managed within the SaaS application itself; they can contain individual user accounts, as well as user groups. Individual user accounts and groups can be assigned several different roles as required.

Depending on the roles to which a user is assigned, he or she is granted one or more permissions to

perform specific operations or actions. These actions typically map directly to important business functions, or to the management of the application itself. For example, a purchasing application might include permissions for creating, submitting, approving, and rejecting purchase orders; an application for mortgage brokers might include permissions for checking a borrower's credit and granting a loan; and so forth. A single permission can be assigned to one or several roles, as necessary; each user will be granted the union of the permissions assigned to all roles to which the user belongs.

Applications can use business rules to control access to actions and resources at a finer level than permissions allow. Business rules introduce conditions that must be satisfied before access is granted. For example, you can use a business rule that allows a user to transfer funds between different accounts only during normal business hours, or if the amount being transferred does not exceed a certain figure.

Access control is managed at the scope level. Each scope inherits roles, permissions, and business rules from any parent scopes, according to the application's relationship strategy, and it can modify, add, and delete them as appropriate. For example, consider a customer based in the United States, with a branch office in Toronto, Canada. The root scope has a role named Benefits Administrator that has a number of permissions related to managing employee benefits, including the administration of the company's 401(k) retirement savings plan. Because 401(k) plans are a creation of U.S. tax law, they are not used in Canada. Therefore, a child scope is created for the Canadian office that inherits the Benefits Administrator role and its permissions, with the exception of the permission that allows the role to modify 401(k) offerings. In place of this permission, the customer adds a permission that allows the role to modify Registered Retirement Savings Plan (RRSP) offerings, the Canadian equivalent of the U.S. 401(k).
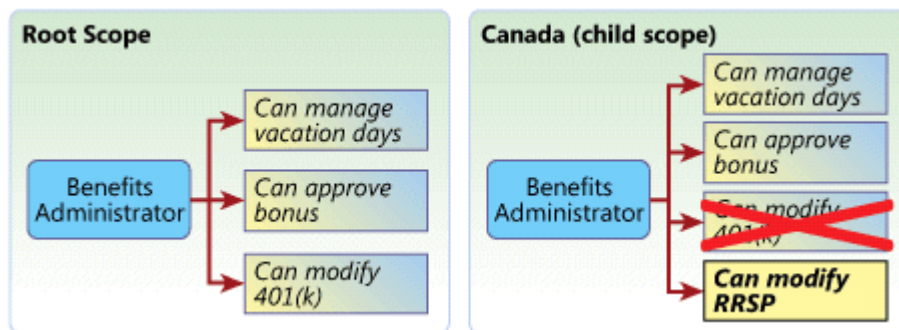
Figure 14. Example of root scope permissions vs. child scope permissions

As a best practice, your application should include a default set of roles, permissions, and business rules that are available to all tenants, and it should allow individual tenants to customize these rules and create more rules through a useful and intuitive user interface.

A Closer Look: Multi-Tenant Data Model

Thus far, we've been covering application architecture at a fairly high level, so let's examine a particular challenge in greater detail: that of creating a data model that customers can extend in a multi-tenant environment. This is by no means a comprehensive exploration of the data model extension process, but it should help give you an idea of the kinds of architectural issues that must be considered when designing SaaS applications.

As designed, your application will naturally include a standard database setup, with default tables, fields, queries, and relationships that are appropriate to the nature of your solution. But different organizations have their own unique needs, which a rigid, inextensible default data model won't be able to address. For example, one customer of a SaaS job-tracking system might have to store an externally-generated

classification code string with each record in order to fully integrate the system with their other processes. A different customer may have no need for a classification string field, but might require support for tracking a category ID number, an integer. Therefore, in all but a few specialized cases, you will have to develop and implement a method by which customers can extend your default data model to meet their needs, without affecting the data model as used by other customers. We'll look at three general approaches to solving this problem: a dedicated tenant database, a shared database with a fixed extension set, and a shared database with custom extensions.

Dedicated Tenant Database

The first approach involves simply giving each tenant its own database, which the tenant can extend as necessary.

With this approach, a new standard default database is created for a new tenant as part of the provisioning process, and the metadata service keeps track of which database is assigned to which tenant. Once the new database is created, the tenant is free to modify it as extensively as your application's user interface and program logic allows, potentially creating new fields, new queries, and even new tables and relationships.

If the cost of providing services is not a factor, this would be the only approach to consider, because it is the simplest arrangement to build, and it offers customers the maximum freedom to extend your default data model. Moreover, customers in fields such as banking or medical records management may have very strong data isolation requirements, and may not even consider an application that does not supply each customer with its own individual database. The disadvantage of this approach is that you will be able to support only a limited number of databases for each server, and therefore your infrastructure cost will be higher, and it will rise more quickly than it would otherwise.

Shared Database, Fixed Extension Set

The second approach involves building a single database that is shared by all of your tenants, and that includes a preset number of custom fields that tenants can assign and use as desired (see Figure 15).

TenantID *Integer*	FirstName *String*	BirthDate *Date*	Custom1 *Integer*	Custom2 *String*	Custom3 *Untyped*
345	Ted	1970-07-02	null	Paid	null
777	Kay	1956-09-25	23	null	null
784	Mary	1962-12-21	null	null	null
345	Ned	1940-03-08	null	Paid	null
438	Pat	1952-11-04	null	San Francisco	Yes

Figure 15. Custom fields in a shared database

In Figure 15, records from different customers are intermingled in a single table; a **TenantID** field associates each record with an individual tenant. In addition to the standard set of fields, a number of custom fields are provided, and each customer can choose what to use these fields for, and how data will be collected for them.

Custom fields can by typed, so that the customer can use any available built-in type checking and verification functions that the application and database provide in order to validate the data. Alternatively, the fields can be untyped, so that the customer can use them to store any type of data. (The customer can optionally provide its own validation logic, to prevent users from accidentally entering invalid data).

A shared database carries a much lower cost of providing services than the isolated approach does, because

it allows a single database engine to support a larger number of customers before partitioning becomes necessary. The biggest disadvantage to this approach is that the extensibility of the data model is limited to the number of custom fields you provide. Choosing this number wisely requires carefully assessing your customers' potential needs. If there are too few custom fields, your customers will not be able to use your application effectively; if there are too many, the result is a sparse, wasteful database with many unused fields.

Shared Database, Custom Extensions

The third approach involves building a single, shared database, and allowing customers to extend the data model arbitrarily, storing custom data as name-value pairs in a separate table (see Figure 16).

TenantID Integer	FirstName String	BirthDate Date	RecordID Integer
345	Ted	1970-07-02	893
777	Kay	1956-09-25	null
784	Mary	1962-12-21	564
345	Ned	1940-03-08	117
438	Pat	1952-11-04	301

RecordID	Name	Value
893	Subscriber	Yes
null	Status	Gold
564	Expire	2008-07-29
117	Subscriber	No
301	Affiliation	Acme
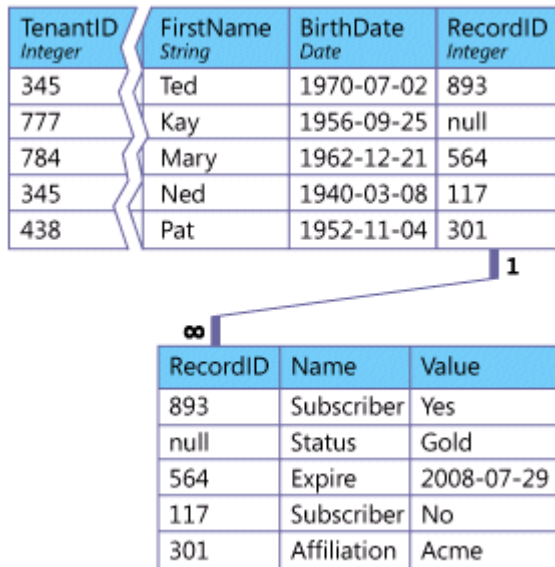
Figure 16. Custom data stored in a separate extension table

Here, each customer record that includes custom data is assigned a unique record ID, which matches one or more rows in a separate extension table. For each row in this table, a name-value pair is stored. Each customer can create as many of these name-value pairs as necessary to meet their business needs. When the application retrieves a customer record, it performs a lookup in the custom data table, selects all rows corresponding to the record ID, and returns them to be treated as ordinary field data. Obviously, data in the custom data table cannot be typed, because it is likely to contain data in many different forms for different customers. To work around this limitation, a third column can optionally hold a data type identifier, so that the data can be cast to the appropriate data type once it is retrieved.

This approach makes the data model arbitrarily extensible, while retaining the cost benefits of using a shared database. The main disadvantage is an added level of complexity for database functions, such as searching, indexing, querying, and updating records. This is typically the best approach to take if you anticipate that your customers will require a considerable degree of flexibility in extending the default data model, but that they won't require data isolation.

When developing an extensibility approach for your data model, remember that any extension implemented by a customer will require a corresponding extension to the business logic (so that the application can use the custom data), as well as an extension to the presentation logic (so that users have a way to enter the custom data as input and receive it as output). The configuration interface that you present to the customer should therefore provide mechanisms for updating all three, preferably in an integrated fashion. (Providing

mechanisms by which customers may extend the business logic and user interface will be addressed in a future paper.)

Scalability

Large-scale enterprise software is intended to be used by thousands of people simultaneously. If you have experience building enterprise applications of this sort, you've gotten to know first-hand the challenges of creating a scalable architecture. For a SaaS application, scalability is even more important: you'll have to support the average user base of a single customer, multiplied by the total number of customers that you have. For ISVs accustomed to building on-premise enterprise software, supporting this kind of user base is like moving from the minor leagues to the majors: the rules may be familiar, but the game is played on an entirely different level. Instead of a widely deployed, business-critical enterprise application, you're really building an Internet-scale system that needs to actively support a user base potentially numbering in the millions.

Scaling the Application

Of course, it's very unlikely that you'll end up supporting as many users as Hotmail does (though if you do, congratulations!). But the scalability challenges are actually quite similar.

Applications can be scaled up (by moving the application to a larger, more powerful server) and scaled out (by running the application on more servers). Scaling up, a familiar solution to anyone who's ever replaced an aging computer with a brand-new model, is often the better choice for smaller applications that don't have to serve very many concurrent users. At the SaaS level, though, scaling out is almost always the best way to add capacity, as depicted in the SaaS maturity model. A well-designed SaaS application can be scaled out to an arbitrarily large number of servers, each running one or more identical instances of the application. The following are some guidelines for designing an application for "scale out":

- Design the application to run in a stateless fashion, with any necessary user and session data stored either on the client side, or in a distributed store that's accessible to any application instance. *Statelessness* means that each transaction can be handled by one instance as well as any other; a user may transact with dozens of different instances during a single session, without ever knowing it.
- Design the application to conduct I/O operations asynchronously, so that the application can perform useful work while waiting for input and output to complete.
- Pool resources such as threads, network connections, and database connections; this helps maximize your computing resources, and it improves your ability to predict resource usage.
- Write your database operations in such a way as to maximize concurrency and minimize exclusive locking. For example, don't lock records when performing read-only operations.

Of course, this is only the very briefest of examinations of the topic; volumes could be (and have been) written about implementing a scalable architecture. For some additional guidance, see the Performance & Scalability resources published by Microsoft Patterns & Practices.

Scaling the Data

As databases serve more users concurrently and grow in size, the amount of time it takes to perform operations such as querying and searching increases significantly. SaaS applications, which often use the same databases to serve thousands of customers, are particularly susceptible to these types of performance degradation, and therefore it's important to plan adequately for growth.

One fairly simple way to scale a database is through partitioning, dividing the data into smaller "chunks" in

order to improve the efficiency of queries and updates. Consider developing a partitioning strategy to determine the best way to partition your data. For example, if an application has customers from around the world, a geographic partitioning strategy might be appropriate, with data belonging to European customers in one partition, data belonging to Asian customers in another, and so on.

In most situations, it is likely that database size will keep growing. Therefore, it is also important to have dynamic repartitioning strategies in place, to ensure that already-partitioned data can be repartitioned in order to keep up with performance and scale metrics.

Operational Structure

The third important shift in thinking has to do with the operational structure of the application: what it takes to deliver the application to customers, and to keep it available and running well at a cost-effective level. For many ISVs, which have never had to run a data center for their customers, this may be the most unfamiliar aspect of SaaS. SaaS providers not only have to be experts in building software and bringing it to market, they must also become experts in operating and managing it.

Resources such as the Microsoft Operations Framework (MOF) provide a great deal of relevant guidance for maintaining system reliability, availability, supportability, and manageability. In addition to the common operation issues that MOF is designed to address, SaaS presents some unique challenges of its own.
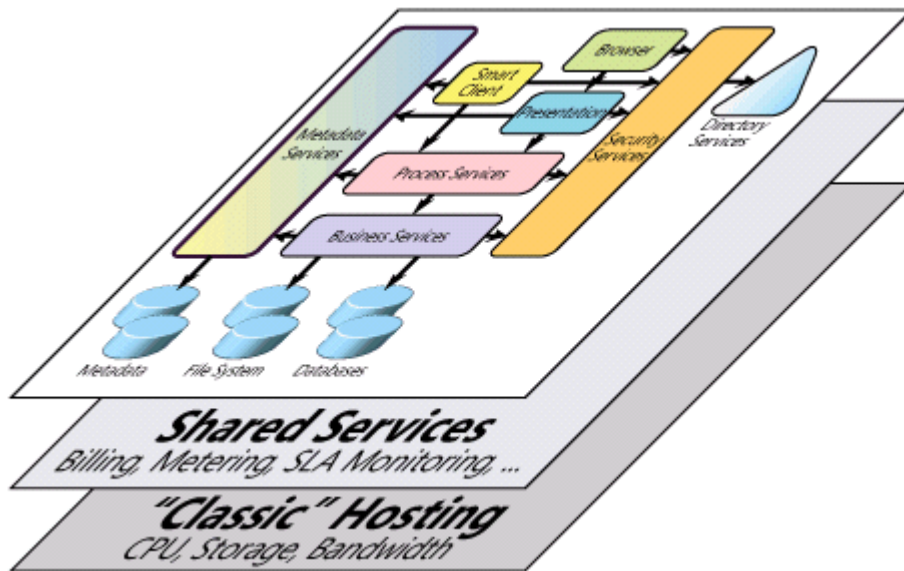
Shared Services

If you've had experience with an enterprise-level World Wide Web presence, you're already familiar with the fundamentals of Web hosting and middleware services, in which an organization either hosts a site internally, or contracts with an external provider for equipment co-location or full-service hosting, including hardware, storage, and network bandwidth. The hosting service is responsible for the availability of the site, but it is typically not otherwise responsible for the site's operation and maintenance.

Providing software as a service adds an additional layer to consider when making hosting arrangements (see Figure 17). Depending on your business plan, you may need a metering and billing system in order to do the following:

- Accurately track customers' usage, and bill them for time or resources used.
- Restrict or throttle access at certain times of the day, or in order to meet other criteria.
- Monitor site access and performance, to ensure that SLAs are being met.
- Perform other functions in order to ensure a seamless experience for your customers that meets or exceeds expectations.

Collectively, the systems used to perform these functions are known as *shared services*.

Figure 17. Shared services layer for SaaS hosting

Shared services can be further classified into two subcategories:

- **Operational support services** (OSS)—Handle operational issues such as account activation, provisioning, service assurance, usage, and metering.
- **Business support services** (BSS)—Support billing (including invoicing, rating, taxation, and collections) and customer management (which includes order entry, customer self services, customer care, trouble ticketing, and customer relationship management).

As with traditional Web hosting, you will need to decide whether to build the shared services layer yourself and self-host your application, or to contract with an external hosting company (known as a *SaaS provider*) to provide it. SaaS providers offer a set of shared services to handle the business and operational issues identified above.

Monitoring

The SLAs that you enter into with your customers will quantify the operational standards that you are required to meet. SLAs are legally binding contracts, and failing to meet them can mean significant lost revenue and damage to your reputation. Monitoring your application architecture for any sign of trouble is therefore a vital tool for detecting problems, and fixing them before they result in significant outages or performance degradation.

Monitoring for Availability

Assuring high availability should be one of the most important priorities for any SaaS vendor. An outage that affects a single server or data center could lead to significant data or productivity losses for a large percentage of your customers—and maybe your entire customer base! For ISVs moving to SaaS from a background in traditional desktop or client–server software development, the high–availability requirements of a net–centric application model can involve new and unfamiliar challenges. It is recommended that you build support for basic techniques, such as heartbeat monitoring and alert mechanisms, into your application, and that you pay special attention to potential weak links, such as a connection to a database at a remote site not under your control.

Of course, technical mechanisms such as alerts are only a part of the process of ensuring high

availability—and if an alert goes off, but nobody responds, it can't really be said to be part of the process at all. Ensure that there are processes in place at your operations center that prescribe specific courses of action, and standards to achieve, in the event of a system failure.

For an overview of the issues surrounding high availability, see "Service Management Functions: Availability" on Microsoft TechNet.

Monitoring for Performance

Your customers expect you to provide them with application access at an acceptable level of performance. To some extent, this expectation will be made explicit by the SLAs that you agree to honor as part of your contract with the customer. Beyond SLAs, however, if customers perceive your application to be slow or unresponsive, they will be more likely to terminate or decline to renew their subscriptions; disgruntled users may make their feelings known on websites and in the pages of industry publications, thus giving your application a negative reputation. Conversely, a fast, lean application that meets users' needs will please customers, and—if they've moved to your software from a less responsive traditional software package—even make them more receptive to SaaS as a category.

To ensure a high level of performance, build support for performance counters into your application directly, if at all possible. Set performance thresholds for metrics such as CPU usage and application response times, and use alerts to notify the appropriate personnel when management events are raised.

Establishing a baseline for performance is generally the most critical activity. With an established baseline, it is much easier to tell when something abnormal is happening, and where the problem is.

Conclusion

There's plenty more to be said about each of the topics addressed in this paper, but hopefully, by this point, you've read enough to begin developing a conceptual framework for understanding SaaS, and how you and your customers may benefit from it. SaaS represents a new paradigm in software delivery, an architectural model built on the principles of multi-tenant efficiency, massive scalability, and metadata-driven configurability to deliver good software inexpensively to existing and potential customers. Adopting these principles now can help put you well on the path to transforming the way you capture the long tail business.

For more information, please see Multi-Tenant Data Architecture

Acknowledgements

Many thanks to Paul Henry for his help with technical writing.

Feedback

The authors gladly welcome your feedback about this paper. Please e-mail all feedback to fredch@microsoft.com or gianpc@microsoft.com . Thank you.