David Chappell
& Associates

INTRODUCING WINDOWS SERVER APPFABRIC

DAVID CHAPPELL

MAY 2010

SPONSORED BY MICROSOFT

CONTENTS

WHAT IS WINDOWS SERVER APPFABRIC?

One of the great truths of building software is this: Application developers shouldn't spend their time creating infrastructure. Even though every application needs some supporting services, the people who write those applications ought to focus solely on creating value for their users. Whatever infrastructure is required should be provided by the platform they're building on.

Given this, one way to improve a platform is to provide better application infrastructure. This is exactly the goal of Windows Server AppFabric. By providing a set of extensions to Windows Server, Microsoft aims at making it easier for Windows developers to create faster, more scalable, and more manageable applications.

The first release of Windows Server AppFabric has two parts:

- AppFabric Caching Services, which can speed up access to frequently accessed information such as session data used by an ASP.NET application.
- AppFabric Hosting Services, making it easier to run and manage services created with Windows Communication Foundation, especially those built using Windows Workflow Foundation.

Windows Server AppFabric is provided as extensions to the Application Server role of Windows Server, and an application is free to use its parts separately or together. This introduction looks at both, describing what each one does and how it can be used.

APPFABRIC CACHING SERVICES

One way to improve the performance and scalability of many applications is to speed up their access to data. With ASP.NET applications, for example, it's easy to make the logic scale: just deploy multiple copies of that logic on multiple servers, then spread user requests across those servers. Improving this aspect of an application's performance can be as easy as adding more servers.

But if all of those servers are relying on a single database server, access to data can quickly become a bottleneck. If an ASP.NET page has to wait to access the data it needs, adding more front-end servers won't help. One option is to run the application's database on a bigger machine. This can work, but there are limits: scaling up with bigger and bigger computers only goes so far, and it can get quite expensive. What's needed is a way to scale out, making frequently accessed data available on multiple computers and thus getting away from a single-server bottleneck.

An effective way to do this is to create a distributed cache that spreads data across multiple machines. Rather than send every request to a single database server, the ASP.NET application can now access the data it needs from one of several different machines. The load is distributed, and the application will likely perform better. This is exactly what's done by AppFabric Caching Services, as described next.

UNDERSTANDING APPFABRIC CACHING SERVICES

The main components of AppFabric Caching Services are a *cache client*, such as an ASP.NET page, that accesses a *cache cluster* containing some number of *cache server* machines. Each cache server runs an instance of AppFabric Caching Services, and each maintains some cached data. Each cache client can also

maintain its own *local cache*, using software provided as part of AppFabric Caching Services. Figure 1 illustrates these components.

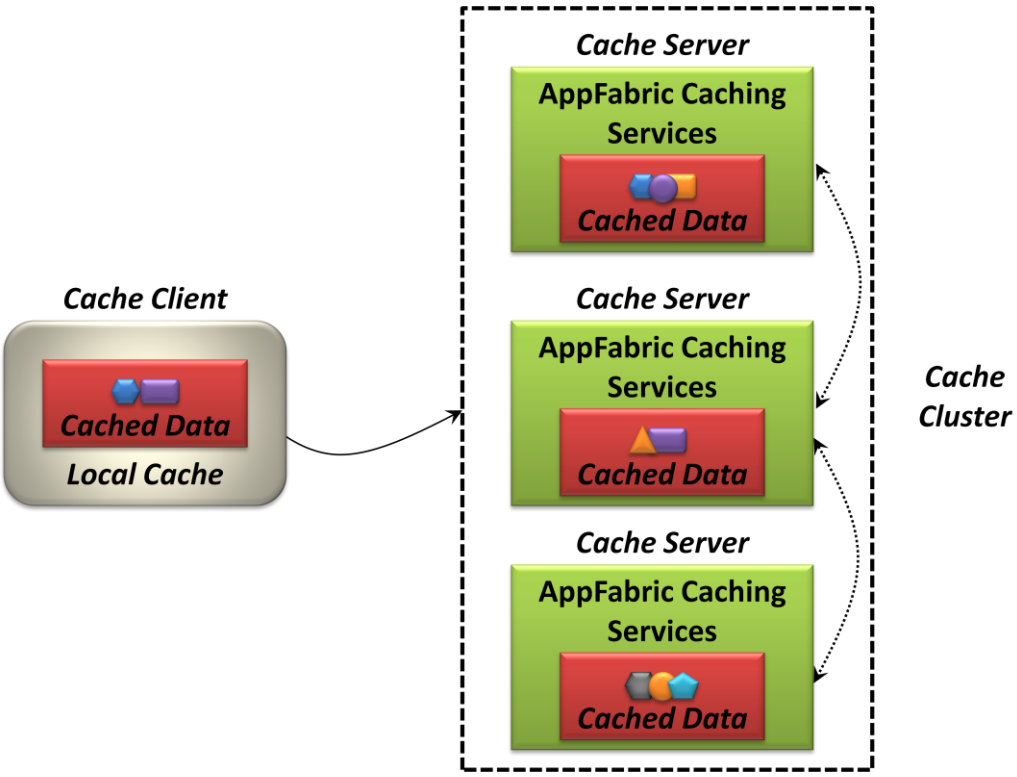

Figure 1: AppFabric Caching Services speeds up access to frequently accessed data.

When the cache client first acquires some item of data, such as information supplied by the user of an ASP.NET application or values read from a database, it can use an AppFabric Caching Services client library to explicitly store this information in the cache cluster under a unique name. (As described later, ASP.NET applications can also do this transparently through the Session object, so using caching needn't require any code changes.) To the client, all of the cache servers in the cluster appear as a single logical store—the client neither knows nor cares which physical server winds up holding the cached data. If it chooses to, the client can also store the data item in its own local cache.

When the client needs to access the same data item again, it asks for it using the item's name. This query first checks the local cache (if one is being used). If the item is found, the client uses this cached value. If the data item isn't in the local cache, the query is then sent to the cache cluster. If the item is found here, the client uses the value returned from the cluster. All of this is transparent to the client—it just requests the item, and AppFabric Caching Services takes care of the rest. If the item isn't found in either the local cache or the cache cluster, the client needs to look elsewhere for the information, such as in the application database.

AppFabric Caching Services was originally code-named "Velocity", a name that suggests exactly what the technology does: It makes repeated access to the same data faster. Rather than forcing an application to make repetitive calls to the database, it instead allows access to data directly from memory, either locally or on one of the cache servers. For applications that frequently access the same data—a large category—caching improves both performance and scalability.

AppFabric Caching Services are designed to be used by .NET applications, and so a cached data item can be any serialized .NET object. Once an object is in the cache, applications can update this cached version or explicitly delete it. Cached data items are also removed by the caching service itself, either through expiration of a configurable time-out period or by being evicted to make room for more frequently accessed information. Data items in the local cache can expire as well, and they can also be set to synchronize automatically with any changes made to the same item in the cache cluster.

Multiple cache clients can share the same cache cluster. This makes sense, since a scalable application will likely replicate its business logic (such as ASP.NET pages) across multiple machines, each of which needs to access the cache. It also raises security issues, however. To make this sharing less risky, all data sent between cache clients and cache servers can be digitally signed and encrypted, and an administrator can limit which accounts have access to each cache. Still, an organization must make sure that all clients using the same cache are trusted, since they can potentially access each other's data.

Caching is useful with various kinds of data. For example, caching makes good sense for unchanging data that's accessed simultaneously by multiple clients, such as catalog information for an online retailer. Another good use of caching is to store data that changes but is accessed by only one client at a time, such as the information in an ASP.NET Session object. Once again, the problem of controlling concurrent access to the cached data doesn't arise.

But what about data that both changes and is accessed simultaneously by multiple clients? Caching can be used here as well, but life gets a little more complicated: Concurrency control is required. To address this situation, AppFabric Caching Services provides both optimistic concurrency control, based on a version number assigned to each cached object, and pessimistic concurrency control, relying on explicit locks.

SCENARIO: USING CACHING IN AN ASP.NET APPLICATION

ASP.NET applications are among the most important clients for AppFabric Caching Services. As just described, the data stored in an ASP.NET Session object is an obvious candidate for caching. In fact, the service provides built-in support for this—the developer just sets a configuration option, and the Session object will be transparently stored in the cache cluster. Notice what this means: ASP.NET developers can get the benefits of caching without changing any of their code. This allows using replicated Web servers, each running an instance of the same ASP.NET application, without either storing session state in SQL Server or requiring sticky sessions. Figure 2 shows how this looks.

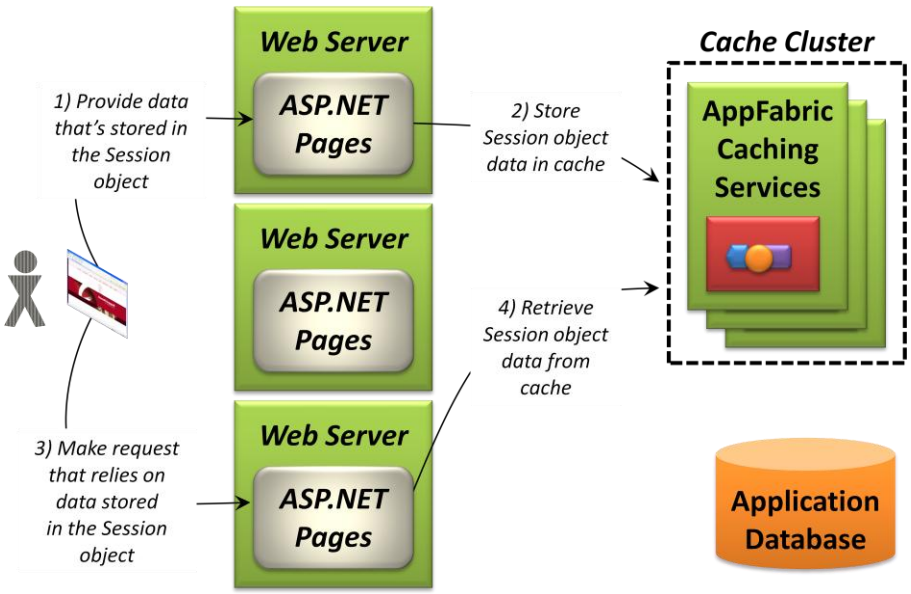

Figure 2: An ASP.NET application can use AppFabric Caching Services to store Session object data.

This simple scenario begins with the user supplying some information that the ASP.NET application stores in the Session object for this user (step 1). The Web server that handles this request has been configured to cache Session objects in an AppFabric Caching Services cluster, and so the user's data is written to one or more cache servers (step 2). The next request from this user relies on the data supplied in step 1, but it's handled by a different Web server (step 3). The ASP.NET code on this Web server refers to the same Session object, which transparently accesses the cache cluster to retrieve the data (step 4). This makes the information provided in step 1 available to the application once again. Notice that there's no need to access the application database here, since the Session object wasn't stored there, nor must the user access the same Web server instance for each request. The result is an ASP.NET application with better performance and improved scalability.

SCENARIO: USING HIGH-AVAILABILITY CACHING

AppFabric Caching Services stores all cached data in memory—it's not written to disk. By default, each cached object is stored on only one machine in a cache cluster. To improve resilience if a cache server goes down, AppFabric Caching Services has a high-availability option that creates a secondary copy of each cached data item on another machine in the cluster. If the cache server holding the primary copy of this data fails, the secondary remains available. Figure 3 illustrates this idea.

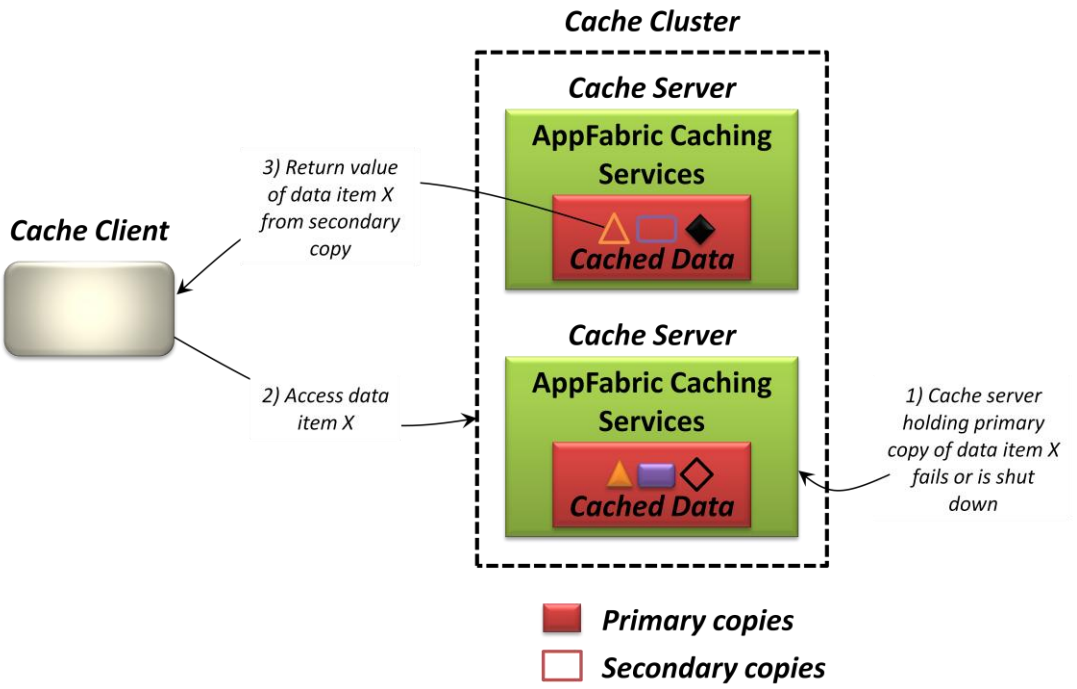

Figure 3: The AppFabric Caching Services high-availability option replicates cached data, letting a secondary copy still be accessed if the primary copy is unavailable.

In this example, the high-availability option is turned on, and so each cached data item is held in two different cache servers. The primary copy, shown in the figure as a filled-in shape, handles all changes to the item. Those changes are automatically propagated to the secondary copy, shown as an open shape. Here, the cache server holding the primary copy of data item X goes down, either intentionally or inadvertently (step 1). When a cache client accesses data item X (step 2), the cache cluster silently redirects that request to the secondary copy and returns its value (step 3).

This example shows a read, but updates also work when the cache server holding a primary copy goes down. Once AppFabric Caching Services detects that the primary is unavailable, it promotes the existing secondary copy to a primary, then creates a new secondary copy. None of this is visible to the cache client—everything works just as if no failure had occurred.

Whether or not the high-availability option is used, AppFabric Caching Services speeds up access to frequently accessed data. This is a good thing—it's a useful addition to the application infrastructure provided by Windows Server. Providing better support for an application's logic is also useful, however. How Windows Server AppFabric does this is described next.

APPFABRIC HOSTING SERVICES

It's become common for applications to expose their functionality through services. On Windows, this most often means implementing those services with Windows Communication Foundation (WCF). And because the logic of some services is best implemented by a workflow, it's also possible to create a WCF service using Windows Workflow Foundation (WF).

But where should these services run? Neither WCF nor WF mandates a particular host process, so developers are free to use them in any way they like. Yet creating an effective, manageable host isn't especially simple. Using both WCF services and workflows would be easier if Windows Server itself provided more support for hosting and managing them.

This is exactly what AppFabric Hosting Services offers. To understand this part of Windows Server AppFabric, however, it's useful to first take a quick look at the fundamental technologies these services support: WCF and WF.

TECHNOLOGY BASICS

WCF provides a general way to expose and consume services. WF offers support for creating business logic as workflows. Since WF workflows typically use services to interact with the outside world, WCF is the place to start in describing these two technologies.

Creating Services: Windows Communication Foundation

WCF lets a developer create and use various kinds of services—SOAP, RESTful, and more—through a common programming model. Figure 4 shows the basics.

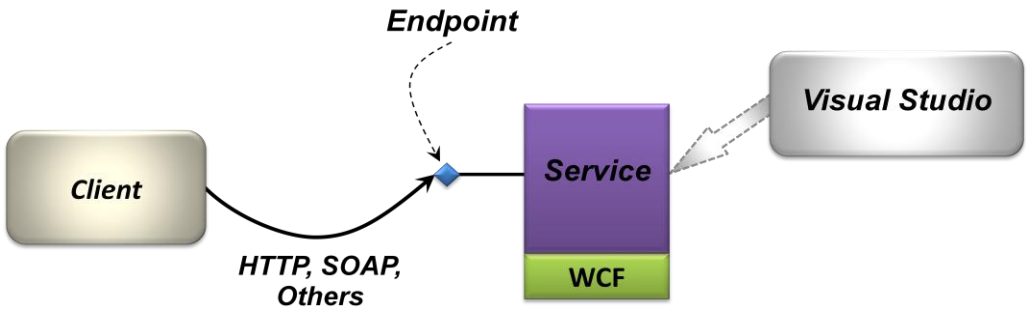

Figure 4: WCF provides a common approach for exposing and consuming services.

Using Visual Studio (or perhaps another tool), a developer can implement a service providing pretty much any function. That service is made available to clients through one or more *endpoints*, each of which exposes a specific *interface* (also referred to as a *contract*).

A service typically divides its functions into some number of *operations*, each carrying out some aspect of its business logic. For example, a service used to implement a shopping application might have operations to create a shopping cart, add an item to the cart, delete an item from the cart, check out, and so on. The service's client then invokes each operation as needed.

Creating Workflows: Windows Workflow Foundation

How should a developer implement a service? In some cases, a service's operations might be entirely independent from one another, such as with a service that allows access to several kinds of data. In other situations, however, the operations a service exposes might carry out some kind of process, which means that they must be called in a specific order over some period of time.

For example, think about how a developer might implement the shopping cart service described earlier, with operations for creating a cart, adding an item, and the rest. The first call a client makes must be to the operation that creates a shopping cart—it's not legal to call anything else before this. Next, the client typically calls the operation to add an item to the cart—calling delete on an empty cart makes no sense—and can then call add and delete as necessary. And once the check-out operation is called, it's no longer legal to invoke operations to add or delete items.

Furthermore, all of this might take place over a potentially long period. Suppose, for instance, that a user creates a shopping cart, adds a couple of items to it, then gets distracted and stops shopping for a while. He might return to this cart hours or days later and expect it still to be available.

How should this kind of logic be implemented? It's a series of steps, with rules defining the order in which those steps must execute. The implementation also must maintain state—the shopping cart—throughout the entire process, which might last for hours, days, or more. Especially if the goal is to create a scalable application, implementing this logic as a workflow built using Windows Workflow Foundation can make sense. WF workflows are designed to make implementing a process easier, and so they can be a good choice for creating some kinds of services.

The basic components of WF are simple: a designer, building blocks for logic, and a runtime engine for implementing workflow-based logic. Figure 5 illustrates these fundamentals.

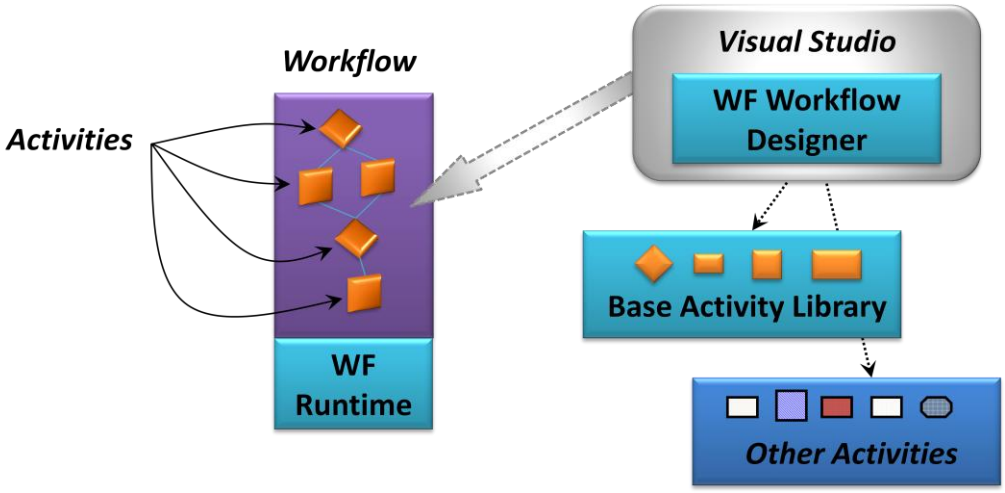

Figure 5: WF is a foundation for implementing process-oriented business logic.

Every WF workflow is made up of *activities*, each of which implements some part of a process's business logic. WF provides a standard *Base Activity Library (BAL)* with activities implementing basic functions such as If and While. Developers are also free to create their own custom activities to carry out whatever business logic is needed. To create a workflow, a developer can use the *WF workflow designer*, part of Visual Studio, to arrange activities on a design surface. Whatever activities it uses, the workflow is executed by a library called the *WF runtime*.

A workflow provides a variety of useful things for a developer implementing a process. For example, when the workflow is waiting for input, such as a request to add or delete a shopping cart item, the WF runtime can automatically persist the workflow's state and unload it from memory. When a new request arrives, the runtime reloads that state, then runs the workflow to handle this request. This makes it easier to build

scalable applications, since the workflow isn't using up memory, threads, or other resources when it's not executing. The runtime can also keep a record of every workflow's execution, something known as *tracking*, letting a developer see things such as when the workflow enters and exits each activity.

Creating a Workflow Service

WF workflows can be used for implementing pretty much any kind of process-oriented logic—they're not limited just to creating services. Still, a WCF service whose logic is created using WF is common enough to have its own name: It's called a *workflow service*. Figure 6 shows how this looks.

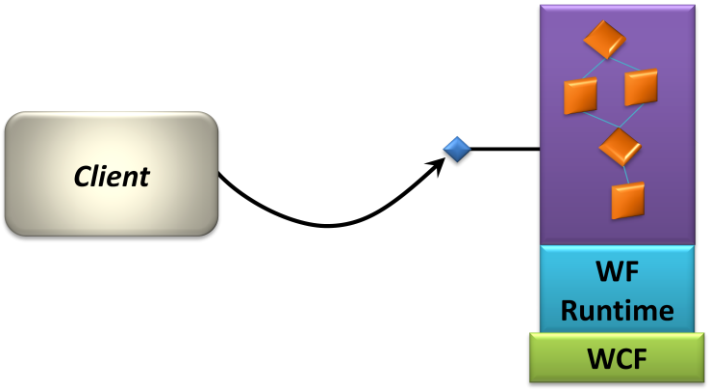

Figure 6: A workflow service is a WCF service whose logic is implemented by a WF workflow.

Suppose a developer creates a workflow service or even an ordinary WCF service, one that doesn't use WF. Neither WCF nor WF defines the host process this service should run in. The good thing about this is that the developer is free to use any process she wishes—WCF and WF don't constrain her. Yet especially for enterprise developers, whose goal is to create business logic, building a host process for a service is just extra work. After all, this process is infrastructure, and providing infrastructure is the responsibility of Windows Server. Helping to meet this need is exactly what AppFabric Hosting Services does, as described next.

UNDERSTANDING APPFABRIC HOSTING SERVICES

AppFabric Hosting Services (originally code-named "Dublin") doesn't create some entirely new hosting infrastructure. Instead, it builds on what's already provided by Internet Information Services (IIS) and Windows Process Activation Service (WAS). On this foundation, AppFabric Hosting Services adds extra capabilities for running and managing WCF services, including workflow services. Figure 7 illustrates this idea.

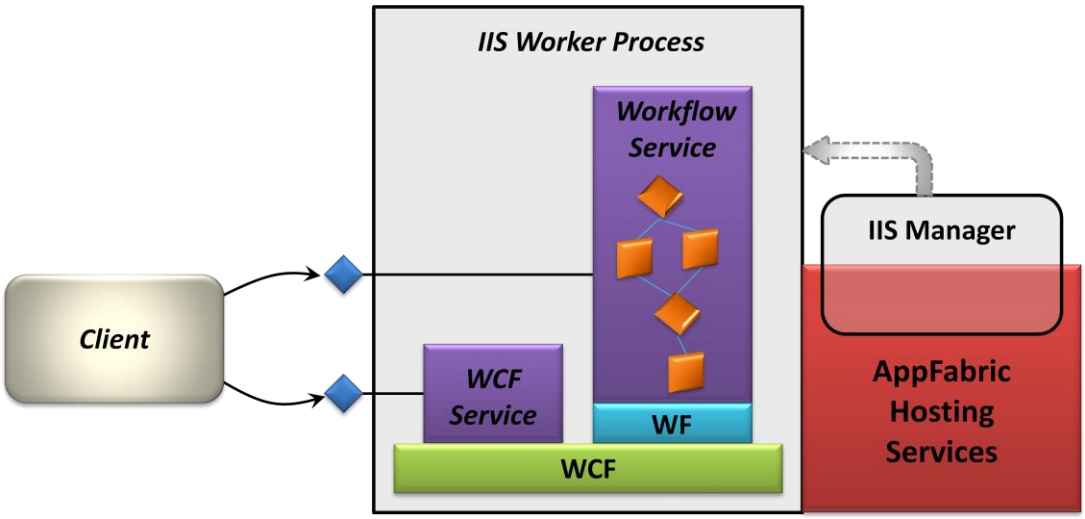

Figure 7: AppFabric Hosting Services make it easier to run and manage WCF services and workflow services.

As the figure shows, WCF services and workflow services run in worker processes provided by IIS—AppFabric Hosting Services doesn't provide its own host process. This technology also takes advantage of WAS, which allows starting a service when a message arrives on either HTTP or another protocol. AppFabric Hosting Services builds on this existing infrastructure, adding things such as the ability to start a particular service when it's deployed rather than waiting for a message to arrive. This helps the service provide a faster response to a client's first request, since there's no need to wait for the service itself to load.

As Figure 7 suggests, AppFabric Hosting Services also extends IIS Manager with new support for managing services. Using these extensions, an administrator can do things such as set WCF configurations, start and stop services, examine their endpoints, and suspend, resume, or terminate specific instances of workflow services. Along with this, AppFabric Hosting Services provides PowerShell cmdlets for service management, letting administrators create custom scripts.

To make life easier for developers, Visual Studio provides built-in project types for creating both WCF services and workflow services. The services created with these project types are immediately deployable in AppFabric Hosting Services—the developer needn't do anything extra. And however they're created, services running in this environment can be used in many different ways, as the scenarios that follow show.

SCENARIO: HOSTING A WORKFLOW SERVICE

While AppFabric Hosting Services can be used with any WCF-based service, it provides extra support for running and managing workflow services. Figure 8 shows some of the most important of these extras.

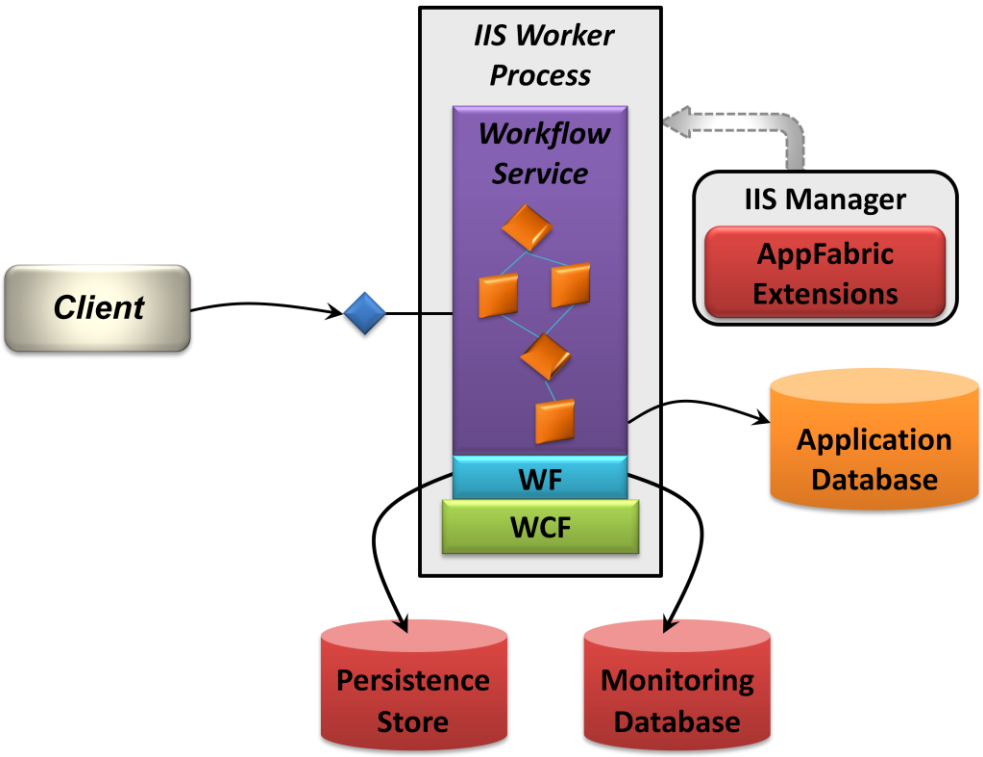

Figure 8: AppFabric Hosting Services provides extra support for running and managing workflow services.

As mentioned earlier, the WF runtime automatically persists the state of a workflow that's waiting for input, then rehydrates it when input arrives. But where is that state persisted? Using WF by itself, it's up to the developer to create and configure a database for this. As Figure 8 shows, however, AppFabric Hosting Services provides a pre-configured persistence store. WF also allows a workflow's execution to be tracked, automatically giving the developer a detailed record of its execution. Once again, though, WF on its own doesn't define where to store that tracking information. As the figure also shows, AppFabric Hosting Services provides a built-in monitoring database. And just to be clear, note that both the persistence store and the monitoring database are distinct from any application database this workflow might be using.

Like any WCF service, workflow services must be monitored and managed. Along with the service management facilities described earlier, the AppFabric extensions to IIS Manager add functions that are specific to workflow services. An example of these, called the AppFabric Dashboard, is shown in Figure 9.

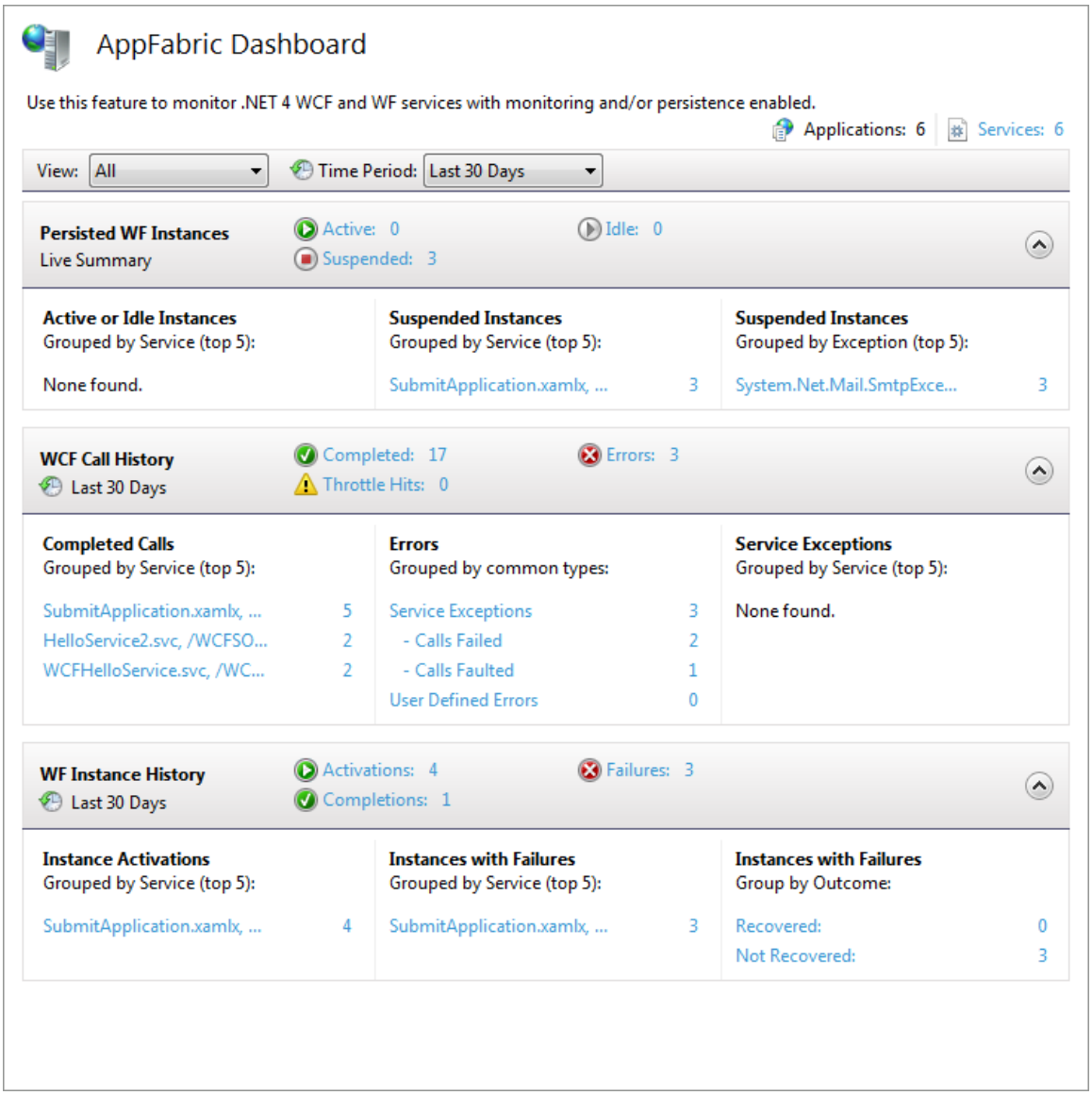

Figure 9: The AppFabric Dashboard shows the current status of services running in AppFabric Hosting Services.

As this example shows, the AppFabric Dashboard provides a window into AppFabric Hosting Services. At the top appears the status of persisted WF instances (i.e., workflow services), showing how many are in each state. Further down, the screen shows a history of recent calls, exceptions, and more. Microsoft also provides a management pack for System Center Operations Manager, allowing events produced by AppFabric Hosting Services to be monitored via this standard Windows management tool. The goal is to give developers and administrators a clear and current view into what's happening in this hosting environment.

SCENARIO: MAKING A WORKFLOW SERVICE MORE SCALABLE

Because it can automatically persist and restore a workflow's state, WF can help developers create scalable business logic. AppFabric Hosting Services lets developers create workflow services that are even more scalable. Figure 10 shows how.

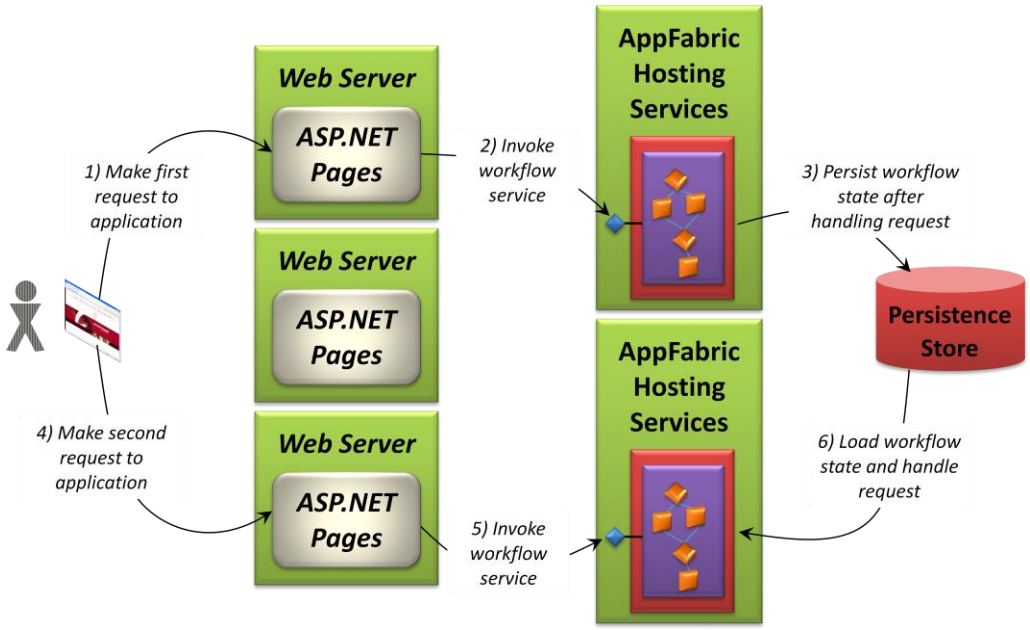

Figure 10: Sequential requests to the same workflow instance can be handled on different machines, with the workflow's state saved and loaded as required.

In this scenario, three Web servers run copies of the same ASP.NET application, with user requests load balanced across them. ASP.NET is used only to handle the user interface, however. This application's logic—maybe it's a shopping cart, as described earlier—is implemented as a workflow service. Two server machines use AppFabric Hosting Services to run instances of this service. (And don't be confused: Even though it's not explicitly shown in the figure, each workflow service runs in an IIS worker process as usual.)

The user's first request to the application is sent to the top Web server in the figure (step 1). The ASP.NET page that gets this request invokes the first operation in the workflow service, such as creating a shopping cart. This request is made to an instance of the service running in the upper of the two middle-tier servers (step 2). Once the operation executes and returns a result, the WF runtime automatically writes the state of the workflow service into the persistence store provided by AppFabric Hosting Services (step 3).

The user's next request is sent by the load balancer to a different Web server machine (step 4). This time, the ASP.NET page that handles the request invokes an operation, such as add item, on an instance of the workflow service running in another of the two middle-tier servers (step 5). Even though this second request is executing on a different machine from the first one, the WF runtime is able to reload the state of this workflow instance from the persistence store and handle the request (step 6).

As this example shows, AppFabric Hosting Services lets the same instance of a workflow service execute on different machines at different times. This makes the service more scalable, since it's now possible to

deploy as many middle-tier servers as required to handle requests. Just as ASP.NET applications can be scaled by adding more Web servers, it's possible to scale business logic implemented as a workflow service by adding more middle-tier servers.

SCENARIO: CONNECTING A WORKFLOW SERVICE TO AN EXISTING APPLICATION USING BIZTALK SERVER

Many applications—maybe even most of them—need to talk with other applications. For example, think once again about a workflow service used to implement an online shopping site. Perhaps the service must submit a completed order to an ERP application to be processed, or maybe it needs to contact another application to verify a credit card. Whatever the requirement, services frequently need to communicate with other software.

In the Microsoft world, this kind of integration is typically handled by BizTalk Server. Figure 11 shows a simple picture of how a workflow service, AppFabric Hosting Services, and BizTalk Server might work together to do this.

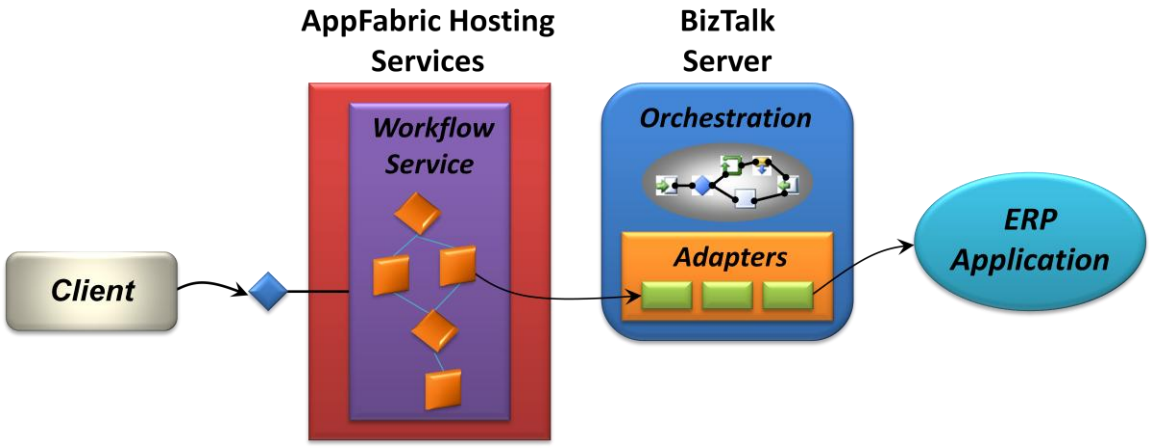

Figure 11: A workflow service running in AppFabric Hosting Services can use BizTalk Server to connect to other applications.

As the figure shows, an activity in the workflow service can issue a request to BizTalk Server. This might be done using a Web service or in some other way; BizTalk Server provides *adapters* for several different communication styles. However it's accomplished, the request can then be handled by business logic running in BizTalk Server. This logic is implemented as an *orchestration*, and it knows how to carry out the next step in this process. Here, for instance, the orchestration uses another BizTalk adapter to communicate directly with the ERP application. Any response can be sent back through the same components.

From a purely technical point of view, a workflow service running in AppFabric Hosting Services can look similar to an orchestration running in BizTalk Server. Both are essentially workflows, both can be created using graphical tools, and Microsoft even provides WCF-based adapters that can be used by both. Don't be confused, however. Despite these similarities, the two technologies address quite distinct problems.

A workflow service running in AppFabric Hosting Services is designed to implement business logic—it's an application. An orchestration running in BizTalk Server, by contrast, is focused entirely on integration. Integration is a specialized problem that requires specialized tools and technologies, and it's exactly what BizTalk Server is built to do. Most of the time, choosing between these two technologies is simple: Use AppFabric Hosting Services to create custom applications, and use BizTalk Server to solve integration problems.

While it might be technically possible to implement some of what BizTalk Server does using WF and AppFabric Hosting Services, doing this probably won't make financial sense. The overall cost of building and maintaining a custom integration server will likely exceed the price of a BizTalk Server license. In fact, it's most accurate to think of AppFabric Hosting Services and BizTalk Server as partners rather than competitors. Each has a specific role to play in creating and connecting applications.

SCENARIO: CONNECTING TO A REMOTE SERVICE THROUGH SERVICE BUS

Another challenge in working with services is exposing them across the Internet. Suppose a WCF service running inside an organization needs to be accessible to clients outside the firewall. Whether or not this service uses WF, there are some roadblocks to doing this. How can client requests get through the firewall, for example? And given that many organizations assign Internet IP addresses dynamically using network address translation (NAT), how can the service expose a fixed IP address to those clients?

Service Bus, part of the Windows Azure platform, was created to address these problems. Figure 12 shows how.

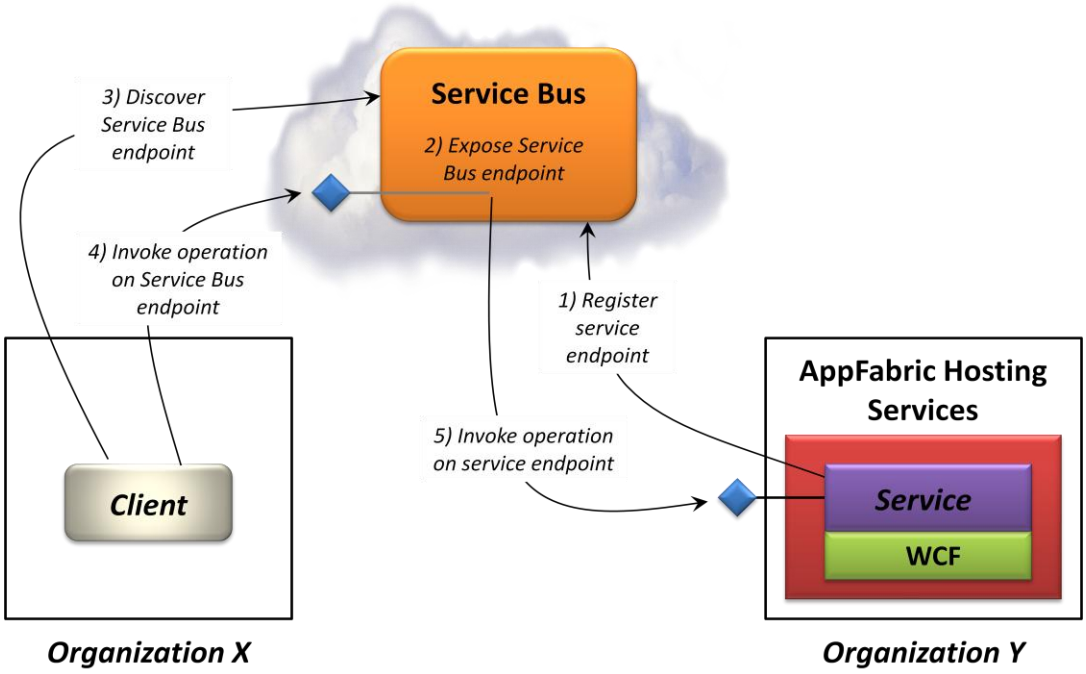

Figure 12: Service Bus makes an on-premises WCF service accessible to clients through the Internet.

To begin, a WCF service registers its endpoint with Service Bus, giving it a unique name (step 1). Service Bus then exposes an endpoint with the same interface (step 2). Using the service's name, a client can

discover the endpoint Service Bus exposes for it (step 3). Once the client knows this, it can invoke operations on the Service Bus endpoint (step 4). For each operation the client calls, Service Bus invokes the corresponding operation in the on-premises service endpoint (step 5).

To understand why this works, it's important to realize that in step 1, when the WCF service registers its endpoint with Service Bus, it establishes a TCP connection with this cloud intermediary. It then leaves this connection open, giving Service Bus a way to invoke operations on the service without being blocked by the firewall. Since the firewall sees those calls as return traffic on a connection that was established from inside the organization, it happily passes them through. Leaving the connection open also gives the WCF service a constant IP address regardless of NAT, letting Service Bus reliably invoke its operations. And since Service Bus itself has a fixed IP address, clients can always find it.

The Windows Azure platform component that contains Service Bus is known as *Windows Azure AppFabric*. Despite its name, this component currently shares no technologies with Windows Server AppFabric. Microsoft says that this will change, however. In particular, both AppFabric Caching Services and AppFabric Hosting Services will find their way into Windows Azure AppFabric in the not-too-distant future. Once this happens, application developers will be able to use the same application infrastructure both on premises with Windows Server and in the cloud with Windows Azure.

SCENARIO: CREATING A COMPOSITE APPLICATION WITH A WORKFLOW SERVICE

As more applications expose their functionality through services, developers are increasingly able to create new software that uses these services. This kind of *composite application* can be useful in a variety of situations. For example, rather than directly customizing an existing ERP application, it might be simpler (and cheaper) to create a composite application that uses ERP services and those of other applications to provide new functionality. As Figure 13 shows, a workflow service running in AppFabric Hosting Services can provide a way to do this.

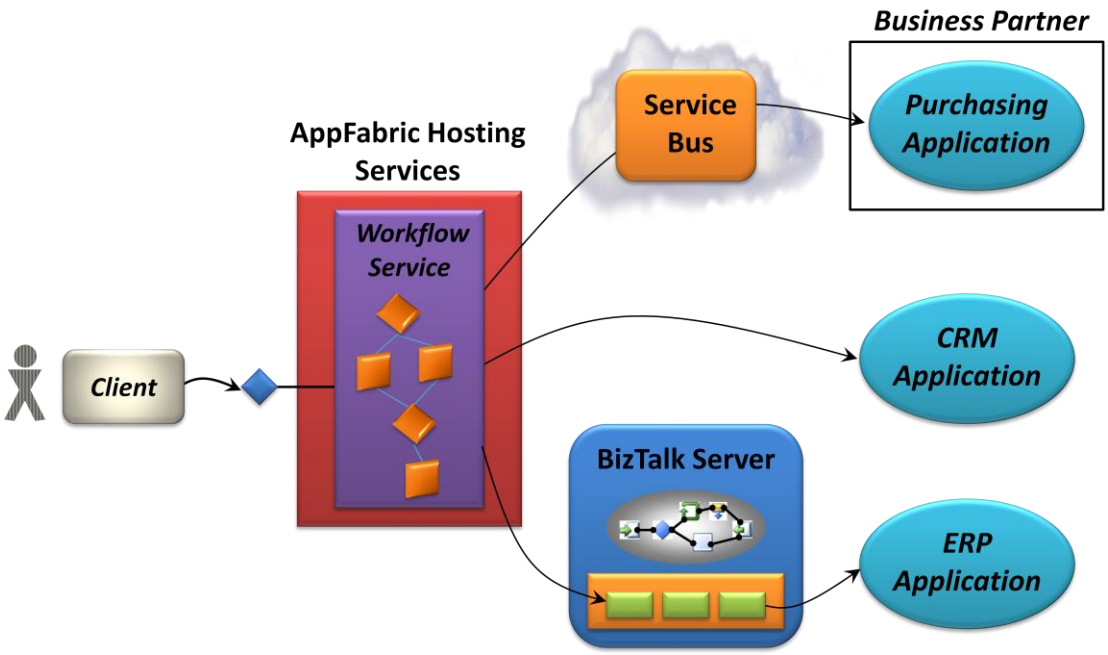

Figure 13: The logic that drives a composite application can be implemented as a workflow service running in AppFabric Hosting Services.

In this example, the workflow service interacts with other applications in several different ways:

- It uses Service Bus to invoke operations in a business partner's purchasing application over the Internet.
- It makes direct calls to a CRM application, perhaps using SOAP-based Web services.
- It relies on BizTalk Server to interact with an ERP application.

In an increasingly service-oriented world, making it as easy as possible to create the logic that drives those services—composite applications—makes sense. AppFabric Hosting Services supports this by providing application infrastructure for workflow services.

PUTTING THE PIECES TOGETHER: COMBINING CACHING AND HOSTING

The two parts of Windows Server AppFabric—AppFabric Caching Services and AppFabric Hosting Services—can each be used independently. It can also make sense to use them together. For example, suppose an organization wishes to create an online shopping application that's able to handle a very high number of simultaneous users. Figure 14 shows how the two components of Windows Server AppFabric could be combined to do this.

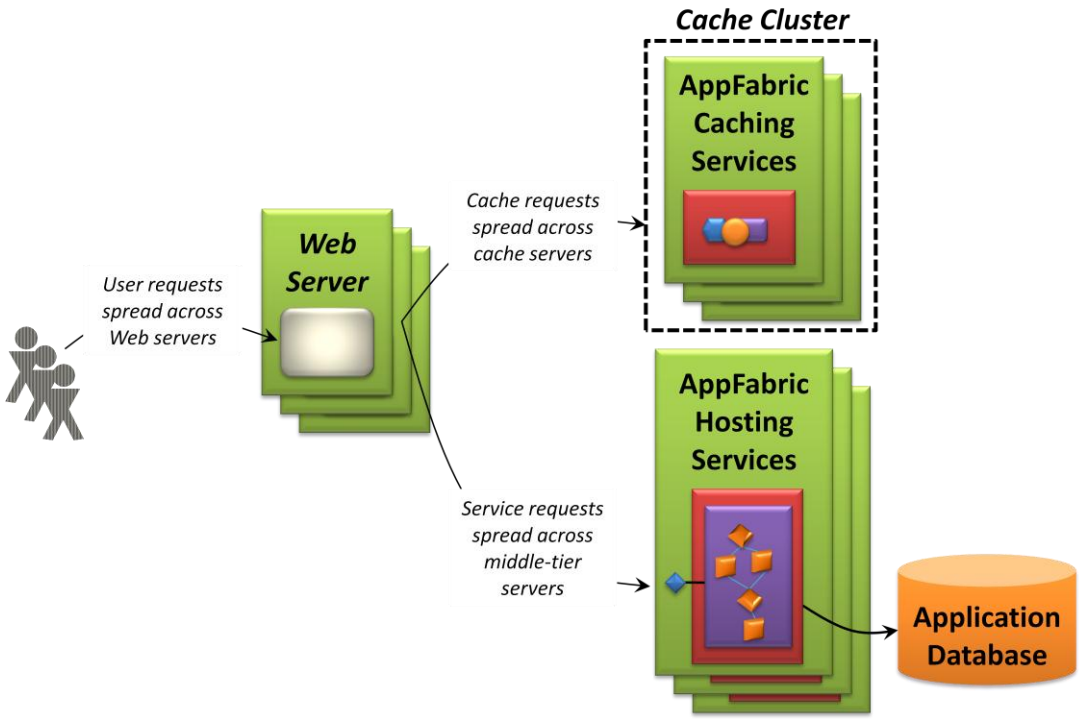

Figure 14: A high-volume Web application might use both AppFabric Hosting Services and AppFabric Caching Services to improve performance, scalability, and manageability.

As always, requests from users can be load-balanced across multiple Web servers, with a copy of the application's business logic—implemented as ASP.NET pages—on each one. If this logic frequently uses the same information, such as catalog data describing what this organization sells, that information can be accessed from a cache cluster implemented using AppFabric Caching Services. And because this application's shopping carts are implemented using workflow services running in AppFabric Hosting Services, requests from Web servers can be spread across multiple middle-tier servers, as described earlier. (Although it's not shown in the figure, the workflows might also access cached data to get product data or other information.) In scenarios like this one, combining both parts of Windows Server AppFabric can make life simpler for developers who need to create very scalable applications.

CONCLUSION

There's no such thing as a good, slow application. There are also no really good applications that are hard to manage. By providing infrastructure to improve the performance, scalability, and manageability of Windows applications, Windows Server AppFabric can help developers create better software.

Like all application infrastructure, this technology is really just plumbing. Yet plumbing is important—it's the foundation for a comfortable life. In the same way, the goal of Windows Server AppFabric is to provide a foundation for creating more effective Windows applications. When developers can spend more time writing business logic and less building infrastructure, everybody is better off.

ABOUT THE AUTHOR

David Chappell is Principal of Chappell & Associates (www.davidchappell.com) in San Francisco, California. Through his speaking, writing, and consulting, he helps people around the world understand, use, and make better decisions about new technology.