Microsoft®

The Workflow Way

Understanding Windows Workflow Foundation

David Chappell, Chappell & Associates

April 2009

Contents

Introducing Windows Workflow Foundation

Everybody who writes code wants to build great software. If that software is a server application, part of being great is scaling well, handling large loads without consuming too many resources. A great application should also be as easy as possible to understand, both for its creators and for the people who maintain it.

Achieving both of these goals isn't easy. Approaches that help applications scale tend to break them apart, dividing their logic into separate chunks that can be hard to understand. Yet writing unified logic that lives in a single executable can make scaling the application all but impossible. What's needed is a way to keep the application's logic unified, making it more understandable, while still letting the application scale.

Achieving this is a primary goal of Windows Workflow Foundation (WF). By supporting logic created using *workflows*, WF provides a foundation for creating unified and scalable applications. Along with this, WF can also simplify other development challenges, such as coordinating parallel work, tracking a program's execution, and more.

WF was first released with the .NET Framework 3.0 in 2006, then updated in the .NET Framework 3.5. These first two versions were useful, especially for independent software vendors (ISVs), but they haven't become mainstream technologies for enterprise developers. With the version of WF that's part of the .NET Framework 4, its creators aim to change this. A major goal of this latest release is to make WF a standard part of the programming toolkit for all .NET developers.

Like any technology, applying WF requires understanding what it is, why it's useful, and when it makes sense to use it. The goal of this overview is to make these things clear. You won't learn how to write WF applications, but you will get a look at what WF offers, why it is the way it is, and how it can improve a developer's life. In other words, you'll begin to understand the workflow way.

The Challenge: Writing Unified, Scalable Application Logic

One simple way to write a program is to create a unified application that runs in a single process on a single machine. Figure 1 illustrates this idea.
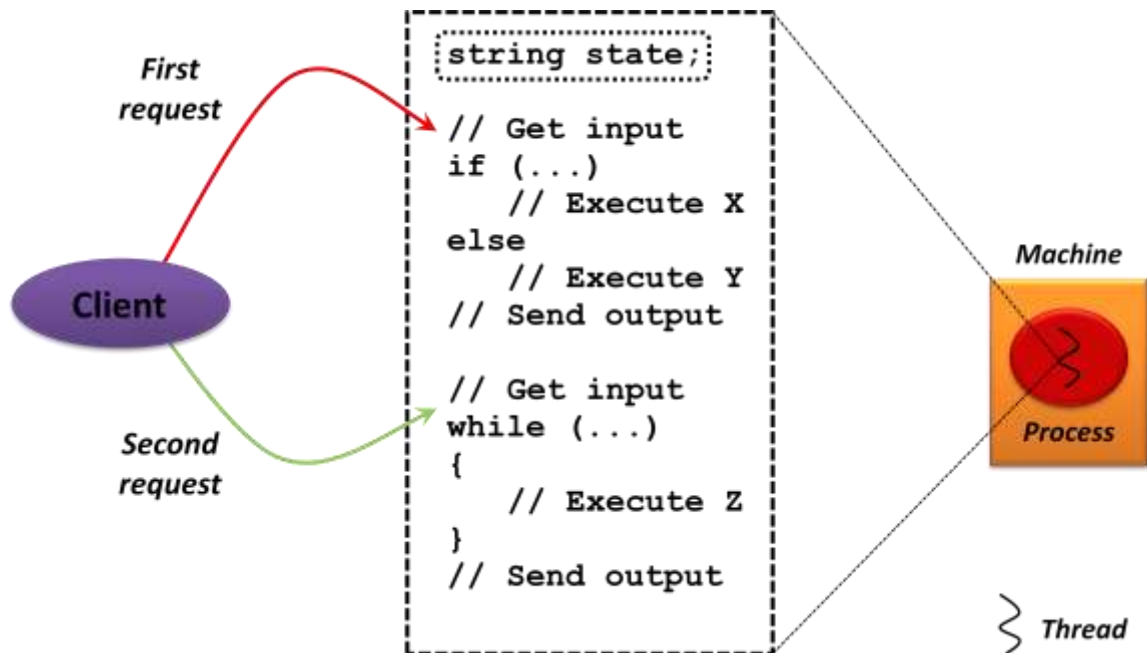
Figure 1: Application logic can be created as a unified whole, then executed on a particular thread in a process running on a single machine.

The simple pseudo-code in this example shows the kinds of things that applications usually do:

☐ Maintain state, which here is represented by a simple string variable.

☐ Get input from the outside world, such as by receiving a request from a client. A simple application could just read from the console, while a more common example might receive an HTTP request from a Web browser or a SOAP message from another application.

☐ Send output to the outside world. Depending on how it's built, the application might do this via HTTP, a SOAP message, by writing to the console, or in some other way.

☐ Provide alternate paths through logic by using control flow statements such as if/else and while.

☐ Do work by executing appropriate code at each point in the application.

In the unified approach shown here, the application logic spends its entire life running on a thread inside a particular process on a single machine. This simple approach has several advantages. For one thing, the logic can be implemented in a straightforward, unified way. This helps people who work with the code understand it, and it also makes the allowed order of events explicit. In Figure 1, for example, it's evident that the client's first request must precede the second—the program's control flow requires

it. When the second request arrives, there's no need to check that the first request has already been handled, since there's no other path through the application.

Another advantage of this approach is that working with the application's state is easy. That state is held in the process's memory, and since the process runs continuously until its work is done, the state is always available: The developer just accesses ordinary variables. What could be simpler?

Still, this basic programming style has limitations. When the application needs to wait for input, whether from a user at the console, a Web services client, or something else, it will typically just block. Both the thread and the process it's using will be held until the input arrives, however long that takes. Since threads and processes are relatively scarce resources, applications that hold on to either one when they're just waiting for input don't scale very well.

A more scalable approach is to shut the application down when it's waiting for input, then restart it when that input arrives. This approach doesn't waste resources, since the application isn't holding on to a thread or a process when it doesn't need them. Doing this also lets the application run in different processes on different machines at different times. Rather than being locked to a single system, as in Figure 1, the application can instead be executed on one of several available machines. This helps scalability, too, since the work can more easily be spread across different computers. Figure 2 shows how this looks.
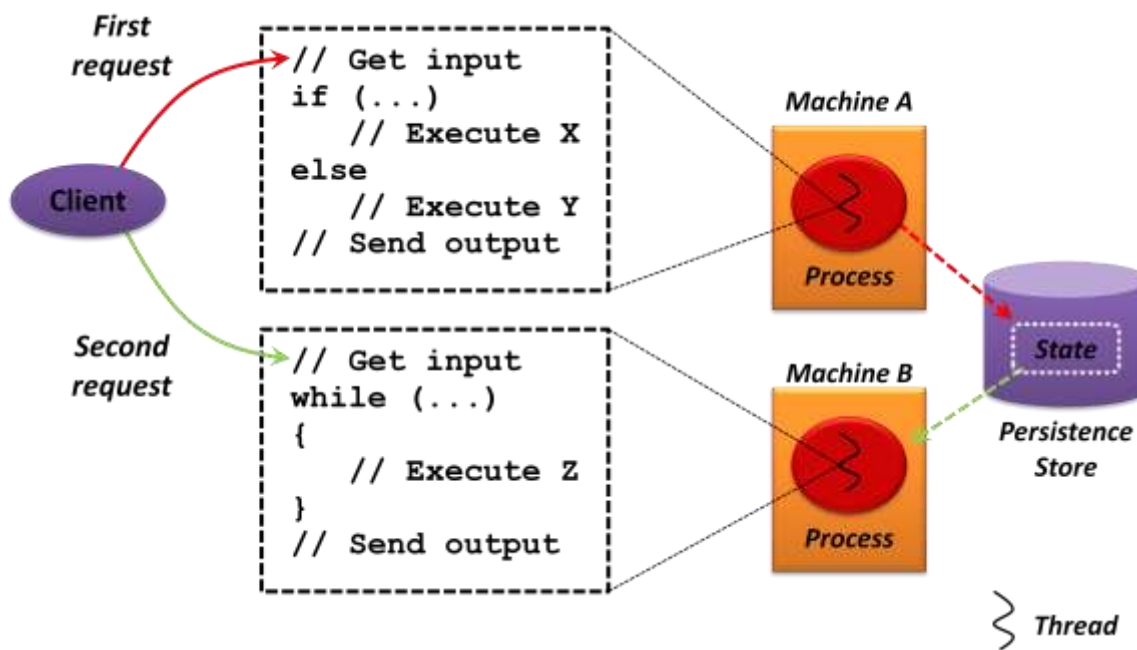

Figure 2: Application logic can be broken into chunks, all sharing common state, that can execute on different machines.

This example application contains the same logic as before, but it's now broken into separate chunks. When the client's first request is received, the appropriate chunk is loaded and executed. Once this request has been handled and a response sent back, this chunk can be unloaded—nothing need remain in memory. When the client's second request arrives, the chunk that handles it is loaded and executed. As Figure 2 shows, this chunk can execute on a different thread in a different process running on a different machine from the first chunk. Once it's handled the request, this second chunk can also be unloaded, freeing whatever memory it was using.

One common example of a technology that works this way is ASP.NET. A developer implements an ASP.NET application as a set of pages, each containing some part of the application's logic. When a request arrives, the correct page is loaded, executed, then unloaded again.

An application built in this style can use machine resources more efficiently than one created using the simpler approach shown earlier, and so it will scale better. Yet we've paid for this improved scalability with complexity. For one thing, the various chunks of code must somehow share state, as Figure 2 shows. Because each chunk is loaded on demand, executed, then shut down, this state must be stored externally, such as in a database or another persistence store. Rather than just accessing ordinary variables, like the scenario shown in Figure 1, a developer now must do special things to acquire and save the application's state. In ASP.NET applications, for instance, developers can write state directly to a database, access the Session object, or use some other approach.

Another cost of this improved scalability is that the code no longer provides a unified view of the program's overall logic. In the version shown in Figure 1, the order in which the program expects input is obvious—there's only one possible path through the code. With Figure 2's version, however, this control flow isn't evident. In fact, the chunk of code that handles the client's second request might need to begin by checking that the first request has already been done. For an application that implements any kind of significant business process, understanding and correctly implementing the control flow across various chunks can be challenging.

The situation is this: Writing unified applications makes the developer's life easy and the code straightforward to understand, but the result doesn't scale well. Writing chunky applications that share external state, such as ASP.NET applications, allows scalability but makes managing state harder and loses the unified control flow. What we'd like is a way to do both: write scalable business logic with simple state management, yet still have a unified view of the application's control flow.

This is exactly what the workflow way provides. The next section shows how.

The Solution: The Workflow Way

Understanding how a WF application solves these problems (and others) requires walking through the basics of how WF works. Along the way, we'll see why this technology can make life better for developers in a surprisingly large number of cases.

Creating Unified Application Logic

A workflow-based application created using WF does the same kinds of things as an ordinary application: It maintains state, gets input from and sends output to the outside world, provides control flow, and executes code that performs the application's work. In a WF workflow, however, all of these things are done by *activities*. Figure 3 shows how this looks, with the unified code approach shown alongside for comparison.
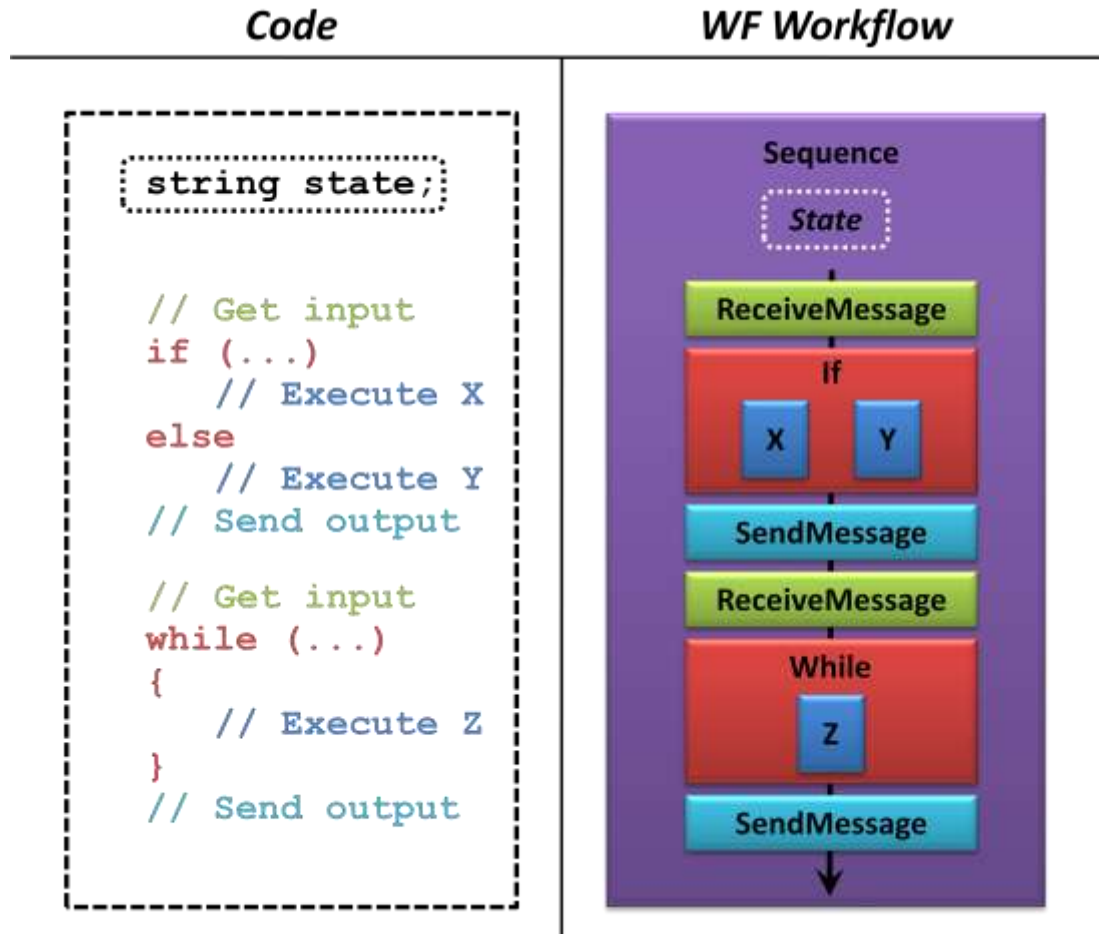
Code	WF Workflow

```
string state;


// Get input
if (...)
    // Execute X
else
    // Execute Y
// Send output

// Get input
while (...)
{
    // Execute Z
}
// Send output
```

Sequence

State

ReceiveMessage

If

X   Y

SendMessage

ReceiveMessage

While

Z

SendMessage

Figure 3: In a WF workflow, all of a program's work is performed by activities.

As Figure 3 shows, every workflow has an outermost activity that contains all of the others. Here, this outermost activity is called Sequence, and like an ordinary program, it can have variables that maintain its state. Because Sequence is a *composite* activity, it can also contain other activities.

In this simple example, the workflow begins with a `ReceiveMessage` activity that gets input from the outside world. This is followed by an `If` activity, which (unsurprisingly) implements a branch. `If` is also a composite activity, containing other activities

(labeled **X** and **Y** here) that execute the work performed on each branch. The **If** activity is followed by a **SendMessage** activity that sends some output to the world beyond this workflow. Another **ReceiveMessage** activity appears next, which gets more input, then is followed by a **While** activity. **While** contains another activity, **Z**, that does the work of this while loop. The entire workflow ends with a final **SendMessage** activity, sending out the program's final result.

All of these activities correspond functionally to various parts of a typical program, as the matching colors in Figure 2 suggest. But rather than using built-in language elements, as a traditional program does, each activity in a WF workflow is actually a class. Execution of the workflow is performed by the *WF runtime*, a component that knows how to run activities. Figure 4 illustrates this idea.
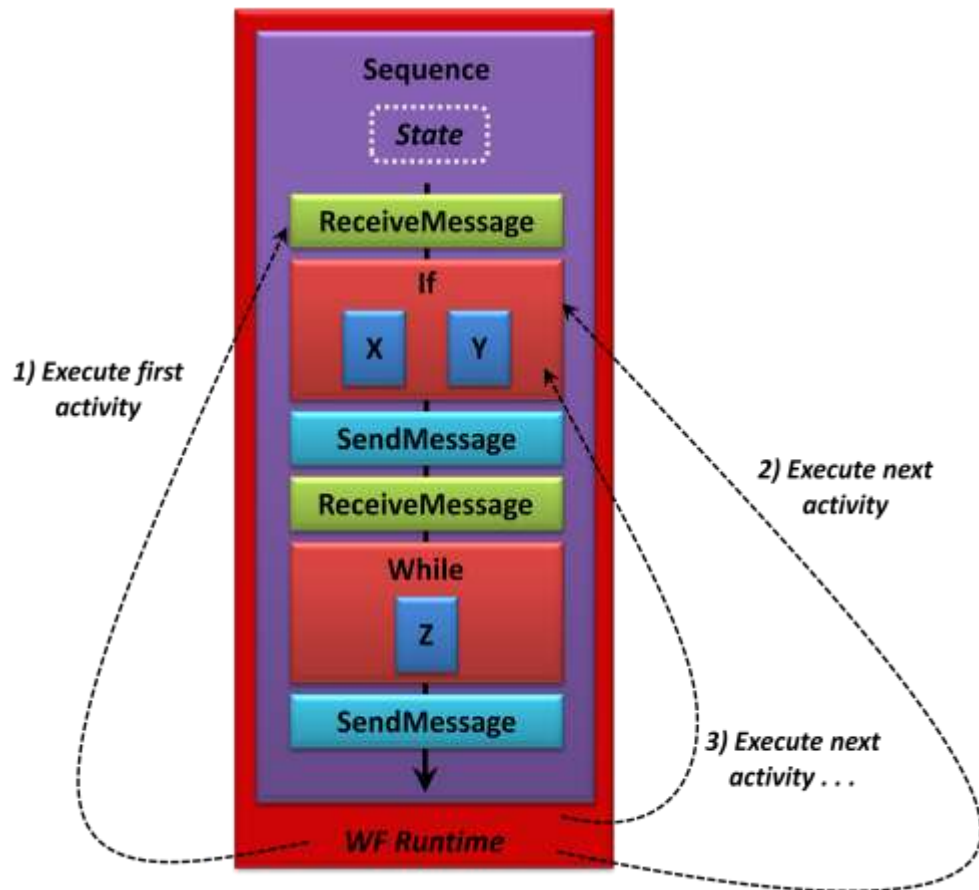

Figure 4: The WF runtime executes activities in the order determined by the workflow.

When it begins running a workflow, the WF runtime first executes the outermost activity, which in this example is a **Sequence**. It then executes the first activity inside that one, which here is **ReceiveMessage**, followed by the next activity, and so on. Exactly which activities are executed in any particular situation depend on what path is taken through the workflow. For example, getting one kind of input in the first

`ReceiveMessage` activity might cause the `If` activity to execute activity `X`, while another kind of input might cause it to execute activity `Y`. It's just like any other program.

It's important to understand that the WF runtime doesn't know anything at all about the internals of the activities it's executing. It can't tell an `If` from a `ReceiveMessage`. The only thing it knows how to do is run an activity, then run the next one. The runtime can see the boundaries between activities, however, which as we'll see is a useful thing.

An important corollary of this is that WF doesn't define any particular language for describing workflows—everything depends on the activities a developer chooses to use. To make life easier, WF includes a *Base Activity Library (BAL)* that provides a broadly useful set of activities. (All of the example activities used here are drawn from the BAL, in fact.) But developers are free to create any other activities they like. They can even choose to ignore the BAL completely.

There's an obvious question here: Why go to all this trouble? Creating a program using activities is different from what developers are used to, so why should anyone bother? Why not just write ordinary code?

The answer, of course, is that this approach can help us create better code. Notice, for example, that the workflow way gives the developer a unified control flow. Just as in the simple case shown back in Figure 1, the program's main logic is defined in one coherent stream. This makes it easier to understand, and because the logic isn't broken into chunks, there's no need for any extra checks. The workflow itself expresses the allowed control flow.

This represents half of our goal: creating unified application logic. But WF applications can also accomplish the second half, creating scalable applications with simple state management. The next section explains how the workflow way makes this possible.

Providing Scalability

To be scalable, a server application can't be locked into a single process on a single machine. Yet explicitly breaking application logic into chunks, as in ASP.NET pages, breaks up what should be a unified control flow. It also forces the programmer to work with state explicitly. What we'd really like is a way to have our logic automatically broken into chunks that can execute in different processes on different machines. We'd also like to have our application's state managed for us, so all we have to do is access variables.

This is exactly what workflows provide. Figure 5 shows the fundamentals of how WF accomplishes these things.
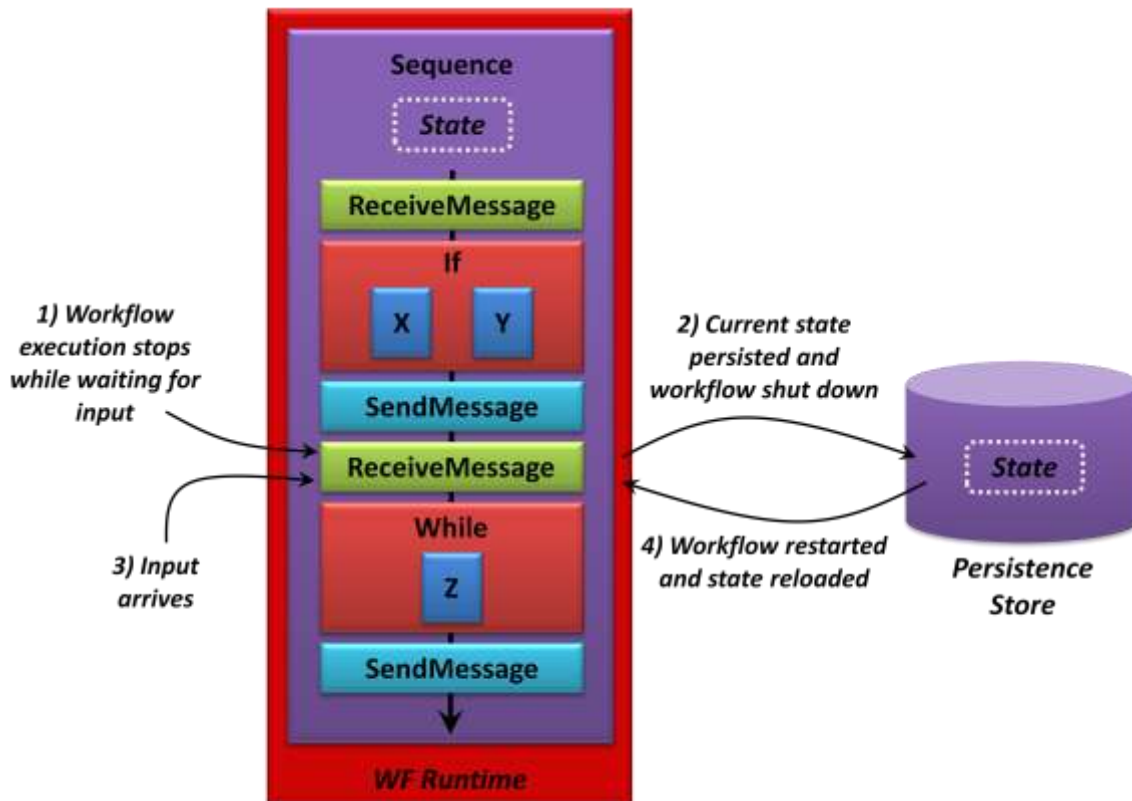
Figure 5: The WF runtime unloads a workflow while it's waiting for input, then loads it again once input arrives.

Like any application, a WF workflow blocks waiting for input. In Figure 5, for example, the workflow is blocked at the second `ReceiveMessage` activity waiting for the client's second request (step 1). The WF runtime notices this, and so it stores the workflow's state and an indication of where the workflow should resume in a persistence store (step 2). When input arrives for this workflow (step 3), the WF runtime finds its persistent state, then reloads the workflow, picking up execution where it left off (step 4). All of this happens automatically—the WF developer doesn't need to do anything. Because the WF runtime can see into the workflow, it can handle all of these details itself.

One obvious advantage of this approach is that the workflow doesn't hang around in memory blocking a thread and using up a process while it's waiting for input. Another advantage is that a persisted workflow can potentially be re-loaded on a machine other than the one it was originally running on. Because of this, different parts of the workflow might end up running on different systems, as Figure 6 shows.
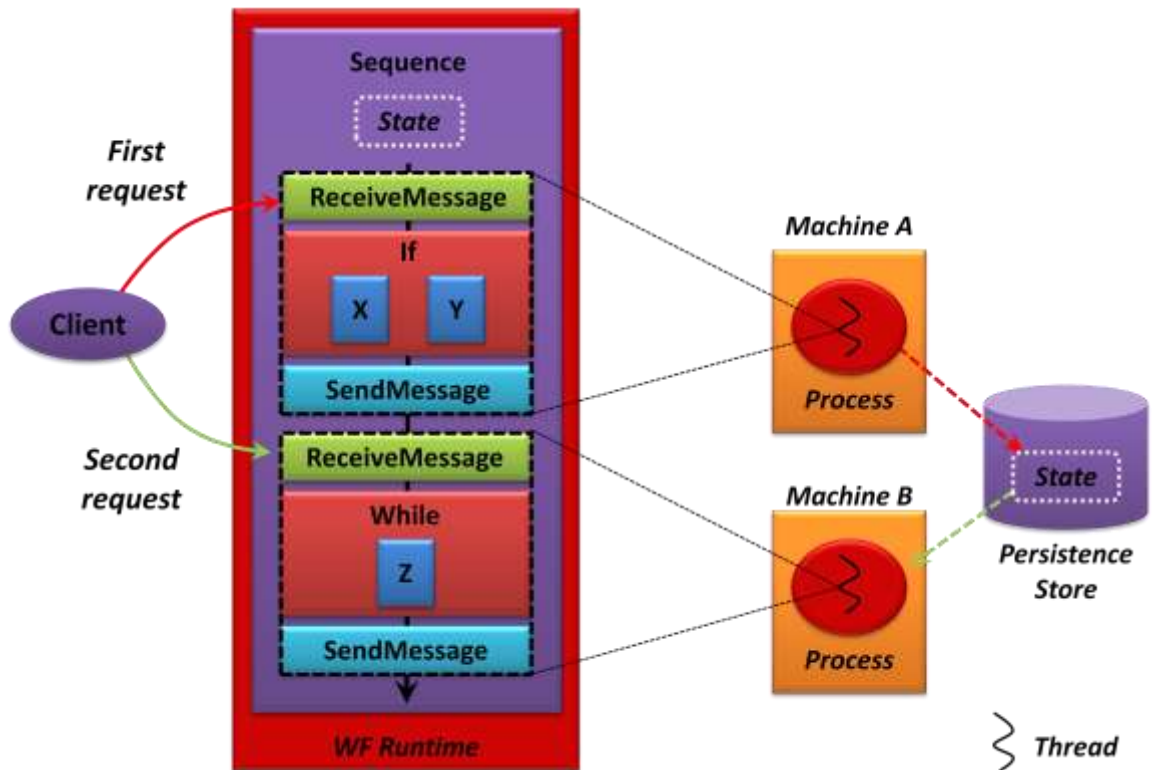
Figure 6: A workflow might run on different threads, in different processes, and on different machines during its lifetime.

In this example, suppose the workflow handles the first client request in a process on machine A. When the second `ReceiveMessage` causes the workflow to block waiting for input, the WF runtime will unload the workflow's state into a persistence store, as just described. When the client's second request arrives, it's possible for this workflow to be reloaded into a process on machine B. Rather than being locked to a specific thread in a specific process on a specific machine, a WF workflow can be moved as necessary. And the developer gets this agility for free—it's provided automatically by WF.

It's worth pointing out that the WF runtime doesn't care how long the workflow has to wait for input. It might arrive a few seconds after the workflow is persisted, a few minutes later, or even a few months later. As long as the persistence store still holds onto the workflow's state, that workflow can be restarted. This makes WF an attractive technology for building applications that implement long-running processes. Think about an application supporting a hiring process, for instance, that encompasses everything from scheduling initial interviews to integrating a new employee into the organization. This process might last weeks or months, and so managing the state of the application using WF makes good sense. Still, while WF can be quite useful with this kind of long-running process, it's important to understand that this isn't its only purpose. Any application that requires unified, scalable logic can be a good candidate for WF.

In fact, take another look at Figure 6. Doesn't it look much like Figure 2, which showed how a chunky application (e.g., one built solely with ASP.NET) achieved scalability? And in fact, doesn't Figure 6 also have a strong similarity to Figure 1, which showed a unified application built with a linear control flow? WF achieves both of these things: The application's control flow is expressed in a comprehensible, unified way, and the application can scale, since it isn't locked to a single process on a single machine. This is the beauty of the workflow way.

And that's not all; using WF also has other advantages. It can make coordinating parallel work easier, for example, help with tracking an application's progress, and more. The next section looks at these aspects of the technology.

Other Benefits of the Workflow Way

A WF workflow consists of activities executed by the WF runtime. While understanding the WF world takes some effort, writing application logic in this style makes a number of common programming challenges easier, as described next.

Coordinating Parallel Work

The WF BAL includes activities that correspond with familiar programming language statements. Because of this, you can write workflows that behave much like ordinary programs. Yet the presence of the WF runtime also allows creating activities that provide more than a conventional programming language. An important example of this is using a workflow to make coordinating parallel work easier.

Don't be confused: The focus here isn't on writing parallel code that runs simultaneously on a multi-core processor. Instead, think about an application that, say, needs to call two Web services, then wait for both results before continuing. Executing the calls in parallel is clearly faster than doing them sequentially, but writing traditional code that does this isn't simple. And while the .NET Framework provides various approaches to making these calls asynchronously, none of them are especially straightforward. This is another situation in which the workflow way can shine: WF makes this easy. Figure 7 shows how.
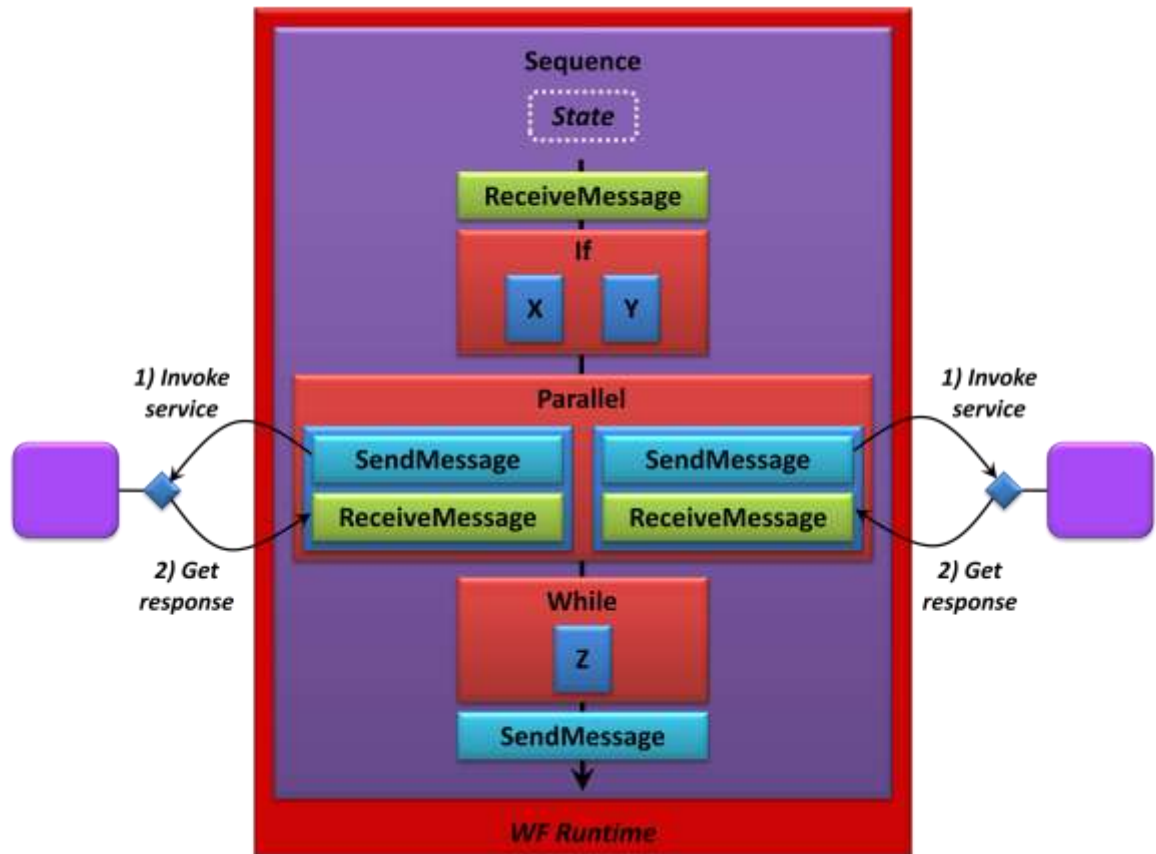
Figure 7: Activities contained within a Parallel activity run in parallel.

This figure shows a simple workflow that's much like the previous example. The big difference is that it now invokes two Web services, then waits for a response from both before continuing. To execute these calls in parallel, the workflow's creator wrapped both pairs of **SendMessage** and **ReceiveMessage** activities inside a **Parallel** activity. **Parallel** is a standard part of WF's Base Activity Library, and it does exactly what its name suggests: executes the activities it contains in parallel. Rather than making the developer deal with the complexity of handling this, the WF runtime and the **Parallel** activity do the heavy lifting. All the developer has to do is arrange the activities as needed to solve her problem.

Providing Automatic Tracking

Since the WF runtime can see the boundaries between a workflow's activities, it knows when each of those activities starts and ends. Given this, providing automatic tracking of the workflow's execution is straightforward. Figure 8 illustrates this idea.
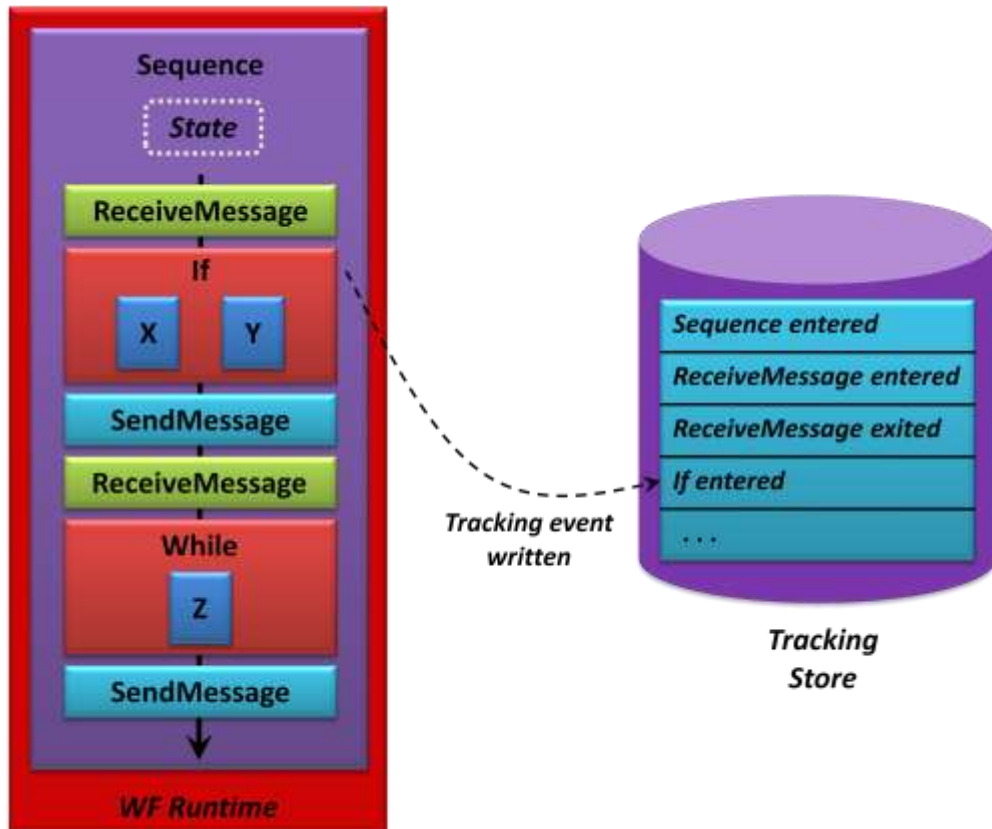
Figure 8: The WF runtime can automatically track a workflow's execution.

As this example shows, the WF runtime can write a record of a workflow's execution into a tracking store. One option is to log activities, with a record written each time an activity begins and ends execution. At the moment shown in the figure, for instance, the workflow is about to begin executing the **If** activity, and so the WF runtime is writing an event indicating this. It's also possible to track specific variables, i.e., workflow state, recording their values at various points in the workflow's execution.

As with other aspects of WF, this automatic logging requires essentially no work on the part of the developer. He can just indicate that tracking should happen, specifying the level he's interested in, and WF takes care of the rest.

Creating Reusable Custom Activities

The activities in WF's BAL provide a variety of useful functions. As already shown, for instance, this built-in set includes activities for control flow, sending and receiving messages, doing work in parallel, and more. But building a real application will usually require creating activities that perform tasks specific to that application.

To make this possible, WF allows creating custom activities. For example, the activities labeled **X**, **Y**, and **Z** in the workflows shown earlier are in fact custom activities, as Figure 9 makes explicit.

Figure 9: Custom activities let a workflow perform application-specific tasks.

Custom activities can be simple, performing just one task, or they can be composite activities containing arbitrarily complex logic. And whether they're simple or complex, custom activities can be used in lots of different ways. For example, a business application created using WF might implement application-specific logic as one or more custom activities. Alternatively, a software vendor using WF might provide a set of custom activities to make its customers' lives easier. For instance, Windows SharePoint Services allows developers to create WF-based applications that interact with people through SharePoint's standard lists. To make this easier, SharePoint includes custom activities for writing information to a list.

Custom activities can be written directly in code, using C# or Visual Basic or another language. They can also be created by combining existing activities, which allows some interesting options. For example, an organization might create lower-level custom activities for its most skilled developers, then package these into higher-level

business functions for less technical people to use. These higher-level activities can implement whatever business logic is required, all neatly wrapped in a reusable box.

Another way to think about this is to view a specific set of activities executed by the WF runtime as a language. If an organization creates a group of custom activities that can be reused to solve specific problems across multiple applications, what they're really doing is creating a kind of domain-specific language (DSL). Once this has been done, it might be possible for less technical people to create WF applications using these pre-packaged chunks of custom functionality. Rather than writing new code to implement the application's functions, useful new software could be created solely by assembling existing activities. This style of DSL, defined in the workflow way, can significantly improve developer productivity in some situations.

Making Processes Visible

Creating applications with a traditional programming language means writing code. Creating applications with WF isn't usually quite so low level. Instead, at least the main control flow of a workflow can be assembled graphically. To allow this, WF includes a workflow designer that runs inside Visual Studio. Figure 10 shows an example.
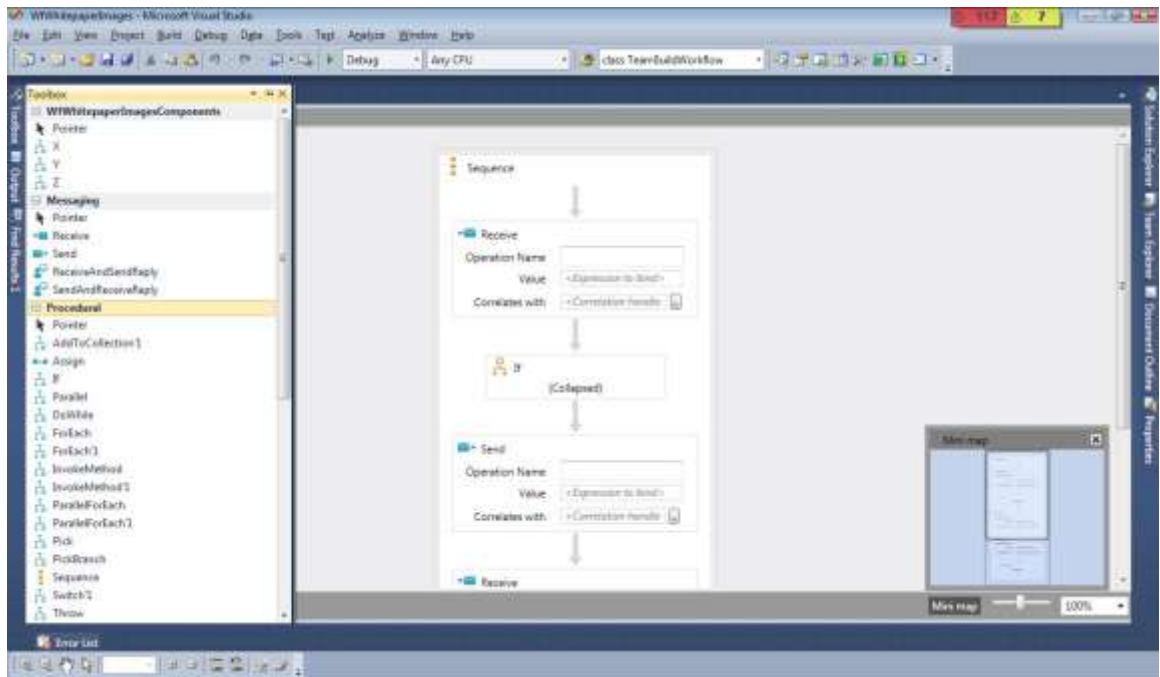
Figure 10: The workflow designer, running inside Visual Studio, lets a developer create a workflow by dragging and dropping activities.

As this example shows, the activities available to a WF developer appear on the left. To create a workflow, she can assemble these activities on the design surface to create whatever logic the application requires. If necessary, the WF workflow designer

can also be re-hosted in other environments, letting others use this tool inside their own offerings.

For some developers, this graphical approach makes creating applications faster and easier. It also makes the application's main logic more visible. By providing a straightforward picture of what's going on, the WF workflow designer can help developers more quickly understand an application's structure. This can be especially helpful for the people who must maintain deployed applications, since learning a new application well enough to change it can be a time-consuming process.

But what about custom activities? Isn't there still some need to write code? The answer is yes, and so WF also allows using Visual Studio to create custom activities in C#, Visual Basic, and other languages. To grasp how this works, it's important to understand how the WF designer represents the various parts of a workflow. Figure 11 shows how this is usually done.
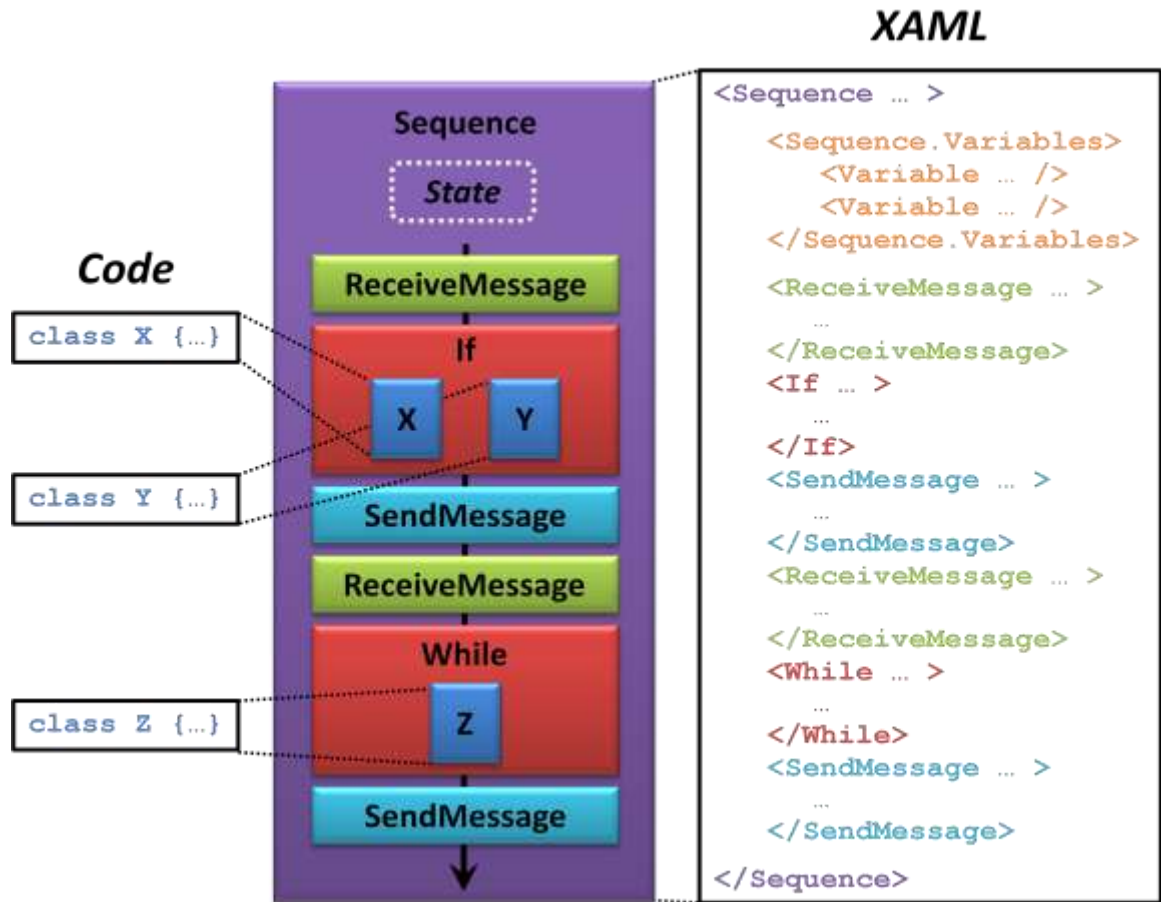

Figure 11: A workflow's state and control flow are typically described in XAML, while custom activities can be written in code.

The composition of a WF workflow—the activities it contains and how those activities are related—is typically represented using the eXtensible Application Markup Language (XAML). As Figure 11 shows, XAML provides an XML-based way to describe the workflow's state as variables and to express the relationships among the workflow's activities. Here, for instance, the XAML for the outermost workflow activity **Sequence** contains a **ReceiveMessage** activity followed by an **If** activity, just as you'd expect.

This **If** activity contains the custom activities **X** and **Y**. Rather than being created in XAML, however, these activities are written as C# classes. While it's possible to create some custom activities solely in XAML, more specialized logic is typically written directly in code. In fact, although it's not the usual approach, a developer is also free to write workflows entirely in code—using XAML isn't strictly required.

Using Windows Workflow Foundation: Some Scenarios

Understanding the mechanics of WF is an essential part of grasping the workflow way. It's not enough, though: You also need to understand how workflows can be used in applications. Accordingly, this section takes an architectural look at how—and why—WF might be used in some typical situations.

Creating a Workflow Service

Building business logic as a *service* often makes sense. Suppose, for example, that the same set of functionality must be accessed from a browser through ASP.NET, from a Silverlight client, and from a standalone desktop application. Implementing this logic as a set of operations that can be invoked from any of these clients—that is, as a service—is likely to be the best approach. Exposing logic as a service also makes it accessible to other logic, which can sometimes make application integration easier. (This is the core idea behind the notion of SOA, service-oriented architecture.)

For .NET developers today, the flagship technology for creating services is Windows Communication Foundation (WCF). Among other things, WCF lets developers implement business logic that's accessible using REST, SOAP, and other communication styles. And for some services, WCF can be all that's needed. If you're implementing a service where each operation stands alone, for example—any operation can be called at any time, with no required ordering—building those services as raw WCF services is just fine. The creator of a service whose operations only expose data might well be able to get away with just using WCF, for instance.

For more complex situations, however, where the operations in a service implement a related set of functionality, WCF alone might not be sufficient. Think about applications that let customers book flight reservations or do online shopping or carry out some other business process. In cases like these, you might well choose to use WF to implement the service's logic. This combination even has a name: Logic implemented using WF and exposed via WCF is known as a *workflow service*. Figure 12 illustrates the idea.
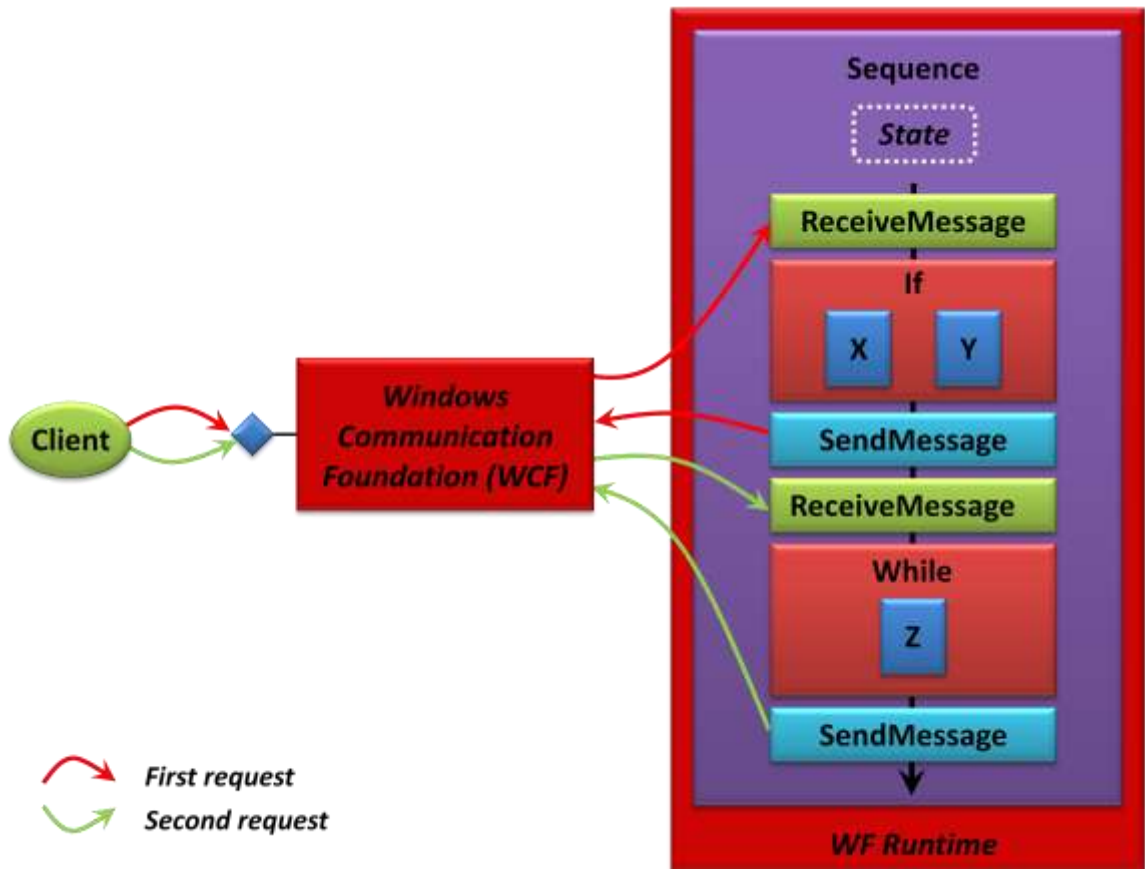
Figure 12: A workflow service uses WCF to expose WF-based logic.

As the figure shows, WCF allows exposing one or more endpoints that clients can invoke via SOAP, REST, or something else. When the client calls the initial operation in this example service, the request is handled by the workflow's first **ReceiveMessage** activity. An **If** activity appears next, and which of its contained custom activities gets executed depends on the contents of the client's request. When the **If** is complete, a response is returned via **SendMessage**. The client's second request, invoking another operation, is handled in a similar way: It's accepted by a **ReceiveMessage**, processed by the workflow's logic, then responded to using a **SendMessage**.

Why is building service-oriented business logic in this way a good idea? The answer is obvious: It allows creating a unified and scalable application. Rather than requiring every operation to contain checks—is it legal to invoke me right now?—this knowledge is embedded in the workflow logic itself. This makes the application easier to write and, just as important, easier to understand for the people who must eventually maintain it. And rather than writing your own code to handle scalability and state management, the WF runtime does these things for you.

A workflow service also gets all of the benefits described earlier, just like any other WF application. These include the following:

- Implementing services that do parallel work is straightforward: just drop activities into a **Parallel** activity.

- Tracking is provided by the runtime.

- Depending on the problem domain, it might be possible to create reusable custom activities for use in other services.

- The workflow can be created graphically, with the process logic directly visible in the WF workflow designer.

Using WF and WCF together like this—creating workflow services—wasn't so easy in earlier versions of WF. With the .NET Framework 4, this technology combination comes together in a more natural way. The goal is to make creating business logic in this style as simple as possible.

Running a Workflow Service with "Dublin"

One big issue with workflows hasn't been discussed yet: Where do they run? The WF runtime is a class, as are activities. All of these things must run in some host process, but what process is this?

The answer is that WF workflows can run in pretty much any process. You're free to create your own host, even replacing some of WF's basic services (like persistence) if you'd like. Plenty of organizations have done this, especially software vendors. Yet building a truly functional host process for WF, complete with management capabilities, isn't the simplest thing in the world. And for organizations that want to spend their money building business logic rather than infrastructure, avoiding this effort makes sense.

A simpler option is to host a WF workflow in a worker process provided by Internet Information Server (IIS). While this works, it provides only a bare-bones solution. To make life easier for WF developers, Microsoft is providing a technology code-named "Dublin". Implemented as extensions to IIS and the Windows Process Activation Service (WAS), a primary goal of "Dublin" is to make IIS and WAS more attractive as a host for workflow services. Figure 13 shows how a workflow service looks when it's running in a "Dublin" environment.
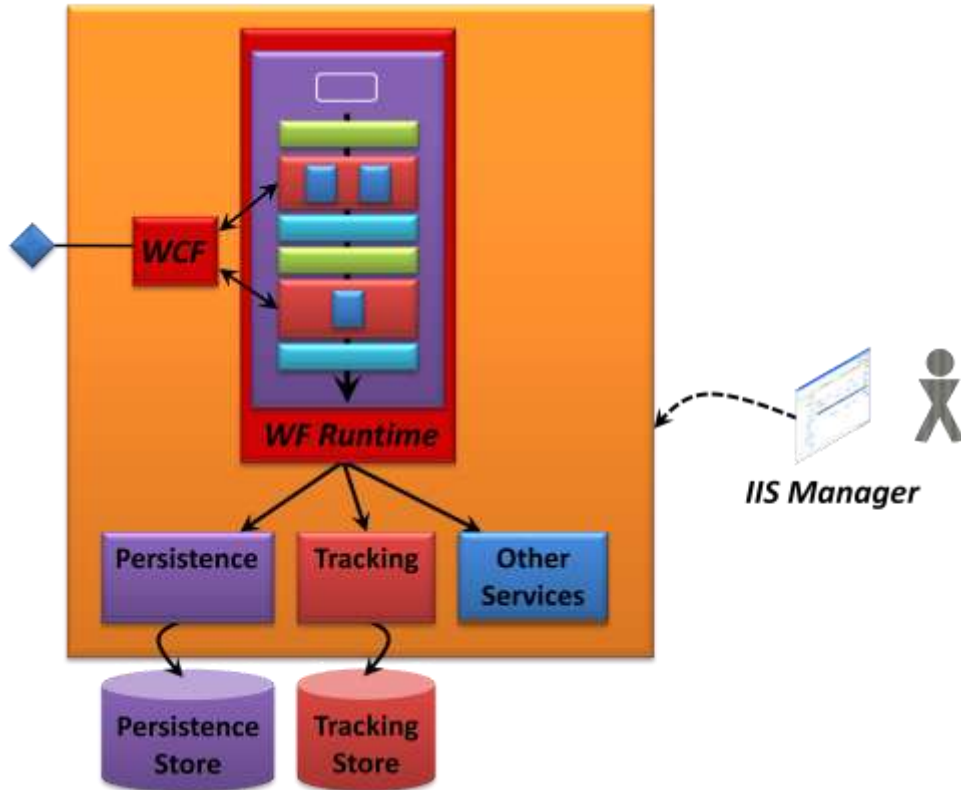
*Figure 13: "Dublin" provides management and more for workflow services.*

While WF itself includes mechanisms for persisting a workflow's state, tracking, and more, it provides only the basics. "Dublin" builds on WF's intrinsic support to offer a more fully useful and manageable environment. For example, along with a SQL Server-based persistence store and a tracking store, "Dublin" provides a management tool for working with these stores. This tool, implemented as extensions to IIS Manager, lets its users examine and manage (e.g., terminate) persisted workflows, control how much tracking is done, examine tracking logs, and more.

Along with its additions to WF's existing functionality, "Dublin" also adds other services. For example, "Dublin" can monitor running workflow instances, automatically restarting any that fail. Microsoft's primary goal with "Dublin" is clear: providing a useful set of tools and infrastructure for managing and monitoring workflow services hosted in IIS/WAS.

Using a Workflow Service in an ASP.NET Application

It's probably fair to say that a majority of .NET applications use ASP.NET. Making the workflow way mainstream, then, means making WF an attractive option for ASP.NET developers.

In earlier versions of WF, workflow performance wasn't good enough for a significant number of ASP.NET applications. For the .NET Framework 4 release, however, WF's designers re-architected important parts of the technology, boosting performance significantly. This release, along with the advent of "Dublin", makes WF-based applications—particularly workflow services—a more viable option for ASP.NET developers. Figure 14 shows a simple picture of how this looks.
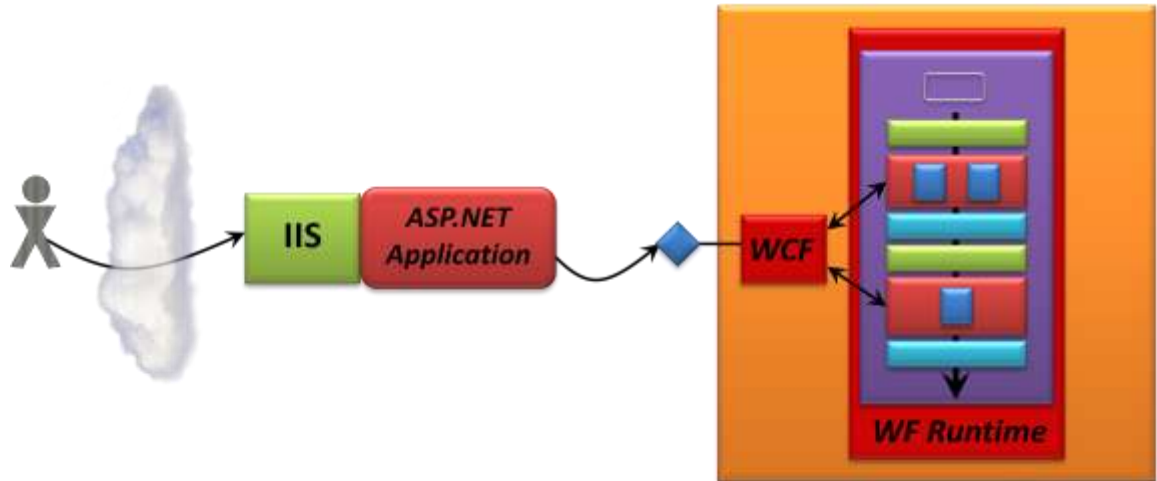
Figure 14: An ASP.NET application's business logic can be implemented as a workflow service

In this scenario, ASP.NET pages implement only the application's user interface. Its logic is implemented entirely in a workflow service. To the workflow service, the ASP.NET application is just another client, while to the ASP.NET application, the workflow service looks like any other service. It needn't be aware that this service is implemented using WF and WCF.

Once again, though, the big question is, Why would you do this? Why not just write the ASP.NET application's logic in the usual way? What does using WF (and WCF) buy you? By this point, most of the answers to those questions are probably obvious, but they're still worth reiterating:

☐ Rather than spreading the application's logic across many different ASP.NET pages, that logic can instead be implemented within a single unified workflow. Especially for big sites, this can make the application significantly easier to build and maintain.

☐ Rather than explicitly working with state in the ASP.NET application, perhaps using the Session object or something else, the developer can rely on the WF runtime to do this. The ASP.NET application need only keep track of an instance identifier for each workflow instance (most likely one per user), such as by storing it in a cookie. When the application supplies this identifier to "Dublin", the workflow instance will be automatically reloaded. It then begins executing where it left off, with all of its state restored.

□   The other benefits of WF also apply, including easier parallelism, built-in tracking, the potential for reusable activities, and the ability to visualize the application's logic.

Because a workflow service isn't tied to a specific process on a specific machine, it can be load balanced across multiple "Dublin" instances. Figure 15 shows an example of how this might look.
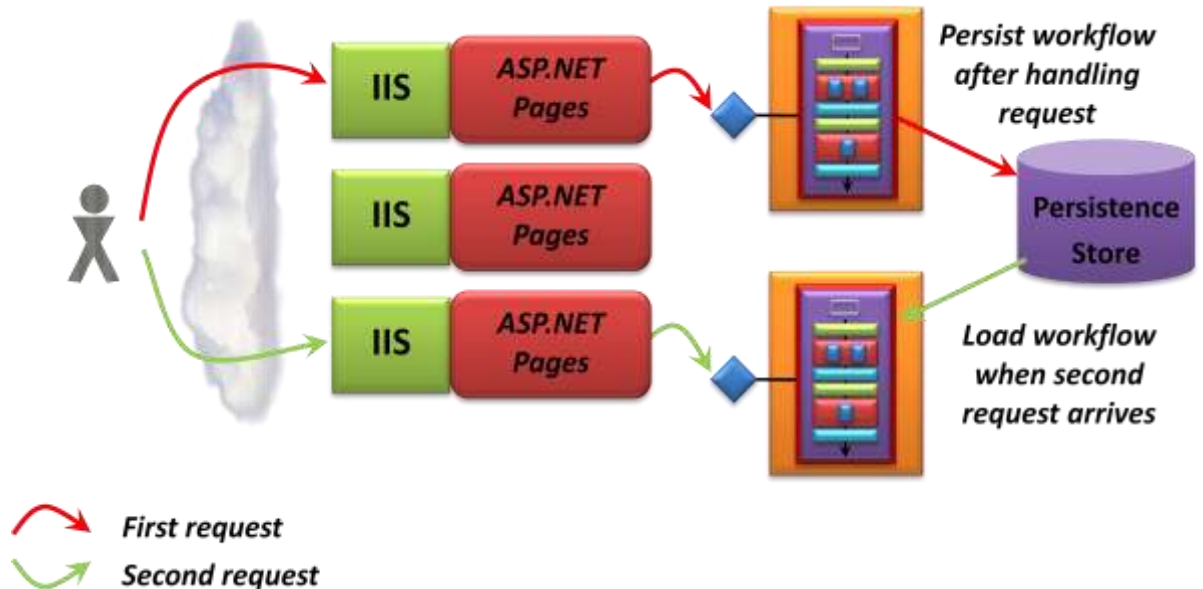

First request

Second request

Figure 15: A replicated ASP.NET application can use multiple "Dublin" instances to execute a workflow service.

In this simple scenario, the pages of an ASP.NET application are replicated across three IIS server machines. While these pages handle interaction with users, the application's business logic is implemented as a workflow service running with "Dublin". Two instances of the "Dublin" environment are running, each on its own machine. When the first request from a user comes in, a load balancer routes it to the top instance of IIS. The ASP.NET page that handles this request calls an operation in the workflow service that implements this chunk of business logic. This causes a WF workflow to begin executing in the "Dublin" instance on the uppermost machine in the figure. Once the relevant activities in this workflow have completed, the call returns to the ASP.NET page, and the workflow is persisted.

The user's second request gets routed to a different instance of IIS. The ASP.NET page on this machine, in turn, invokes an operation in the (persisted) workflow on a different machine from the one that handled her first request. This is no problem: "Dublin" just loads the workflow instance from the persistence store it shares with its fellow server. This reloaded workflow service then handles the user's request, picking up where it left off.

23

Understanding WF (and WCF), then learning to create logic in this new style takes some work. For simple ASP.NET applications, climbing this learning curve might not be worth the effort required. Anybody creating larger Web applications using .NET, however, should at least consider the possibility of creating their logic as a workflow service.

Using Workflows in Client Applications

The focus so far has been entirely on using WF to create server applications. Yet while this is how the technology is most often used today, WF can also be helpful for client applications. Its scalability aspects don't typically add much in this case, since code that runs on client machines usually doesn't need to handle lots of simultaneous users, but WF can still be of use.

For example, think about a client application that presents its user with a graphical interface created using Windows Forms or Windows Presentation Foundation. The application will probably have event handlers to process the user's mouse clicks, perhaps spreading its business logic across these event handlers. This is much like an ASP.NET application spreading its logic across a group of separate pages, and it can create the same challenges. The flow of the application could be hard to discern, for instance, and the developer might need to insert checks to make sure that things are done in the correct order. Just as with an ASP.NET application, implementing this logic as a unified WF workflow can help address these concerns.

Other aspects of WF can also be useful on the client. Suppose the application needs to invoke multiple back-end Web services in parallel, for example, or can benefit from tracking. Just as on the server, WF can help address those challenges. While a majority of WF applications today run on servers, it's important to recognize that this isn't the only choice.

A Closer Look: The Technology of Windows Workflow Foundation

The goal of this overview is not to make you a WF developer. Still, knowing just a little more about the technology of WF can help in deciding when it makes sense to choose the workflow way. Accordingly, this section takes a closer look at some of WF's most important parts.

Kinds of Workflows

In the .NET Framework 4, WF developers typically choose between two different styles of workflow by choosing different outermost activities. Those activities are:

☐ **Sequence**: Executes activities in sequence, one after another. The sequence can contain **If** activities, **While** activities, and other kinds of control flow. It's not possible to go backwards, however—execution must always move forward.

☐ **Flowchart**: Executes activities one after another, like a **Sequence**, but also allows control to return to an earlier step. This more flexible approach, new in the .NET Framework 4 release of WF, is closer both to how real processes work and to the way most of us think.

While both **Sequence** and **Flowchart** can act as the outermost activities in a workflow, they can also be used within a workflow. This lets these composite activities be nested in arbitrary ways.

In its first two releases, WF also included another option for a workflow's outermost activity called **State Machine**. As its name suggests, this activity let a developer explicitly create a state machine, then have external events trigger activities in that state machine. WF in the .NET Framework 4 is a big change, however, one that required re-writing most of the activities in the earlier releases and building a new designer. Because of the effort involved, WF's creators will not be delivering the **State Machine** activity from the initial .NET Framework 4 release. (Even Microsoft's resources aren't without limit.) Still, the new **Flowchart** activity should address many of the situations that previously required using **State Machine**.

The Base Activity Library

A workflow can contain any set of activities a developer chooses to use. It's entirely legal, for example, for a workflow to contain nothing but custom activities. Still, this isn't very likely. Most of the time, a WF workflow will use at least some of what's provided in the Base Activity Library. Among the more broadly useful BAL activities provided by WF in the .NET Framework 4 are the following:

☐ **Assign**: assigns a value to a variable in the workflow.

☐ **Compensate**: provides a way to do compensation, such as handling a problem that occurs in a long-running transaction.

☐ **DoWhile**: executes an activity, then checks a condition. The activity will be executed over and over as long as the condition is true.

☐ **Flowchart**: groups together a set of activities that are executed sequentially, but also allows control to return to an earlier step.

☐ **ForEach**: executes an activity for each object in a collection.

☐ **If**: creates a branch of execution.

☐ **Parallel**: runs multiple activities at the same time.

☐ **Persist**: explicitly requests the WF runtime to persist the workflow.

☐ **Pick**: allows waiting for a set of events, then executing only the activity associated with the first event to occur.

- **ReceiveMessage**: receives a message via WCF.

- **SendMessage**: sends a message via WCF.

- **Sequence**: groups together a set of activities that are executed sequentially. Along with acting as a workflow's outermost activity, **Sequence** is also useful inside workflows. For example, a **While** activity can contain only one other activity. If that activity is **Sequence**, a developer can execute an arbitrary number of activities within the while loop.

- **Switch**: provides a multi-way branch of execution.

- **Throw**: raises an exception.

- **TryCatch**: allows creating a try/catch block to handle exceptions.

- **While**: executes a single activity as long as a condition as true.

This isn't a complete list—WF in the .NET Framework 4 includes more. Also, Microsoft plans to make new WF activities available on CodePlex, its hosting site for open source projects. Shortly after the .NET Framework 4 release, for example, look for activities that let workflows issue database queries and updates, execute PowerShell commands, and more.

As this list suggests, the BAL largely echoes the functionality of a traditional general-purpose programming language. This shouldn't be surprising, since WF is meant to be a general-purpose technology. As this overview has described, however, the approach it takes—activities executed by the WF runtime—can sometimes make life better for application developers.

Workflow in the .NET Framework 4

Release 4 of the .NET Framework brings significant changes to Windows Workflow Foundation. The activities in the BAL have been re-written, for example, and some new ones have been added. This brings real benefits—many workflows now run much faster, for instance—but it also means that workflows created using earlier versions of WF can't be executed by the .NET 4 version of the WF runtime. These older workflows can still run alongside .NET Framework 4 workflows unchanged, however—they needn't be thrown away. Also, activities created using older versions of WF, including entire workflows, can potentially run inside a new **Interop** activity that WF in the .NET Framework 4 provides. This lets logic from older workflows be used in new ones.

Along with better performance, this new release of WF also brings other interesting changes. For example, earlier versions of WF included a **Code** activity that could contain arbitrary code. This let a developer add pretty much any desired functionality to a workflow, but it was a slightly inelegant solution. In the .NET Framework 4, WF's creators made writing custom activities significantly simpler, and so the **Code** activity has been removed. New functionality is now created as a custom activity.

The changes in the .NET Framework 4 release are the most substantial since the original appearance of WF in 2006. All of them have the same goal, however: making it easier for developers to build effective applications using workflows.

Conclusion

Windows Workflow Foundation offers real advantages for many applications. In its first releases, WF struck a chord mostly with software vendors. These original incarnations of the technology were useful, but they weren't really appropriate for mainstream enterprise use. With the .NET Framework 4, the creators of WF are looking to change this.

By making WF and WCF work well together, improving WF's performance, providing a better workflow designer, and offering a full-featured host process with "Dublin", Microsoft is making the workflow way more attractive for a broader range of scenarios. Its days as a specialized player in the Windows application development drama are drawing to a close. WF is on its way to center stage.

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.