Realtime publishers

Solutions for Automating IT Job Scheduling

Greg Shields

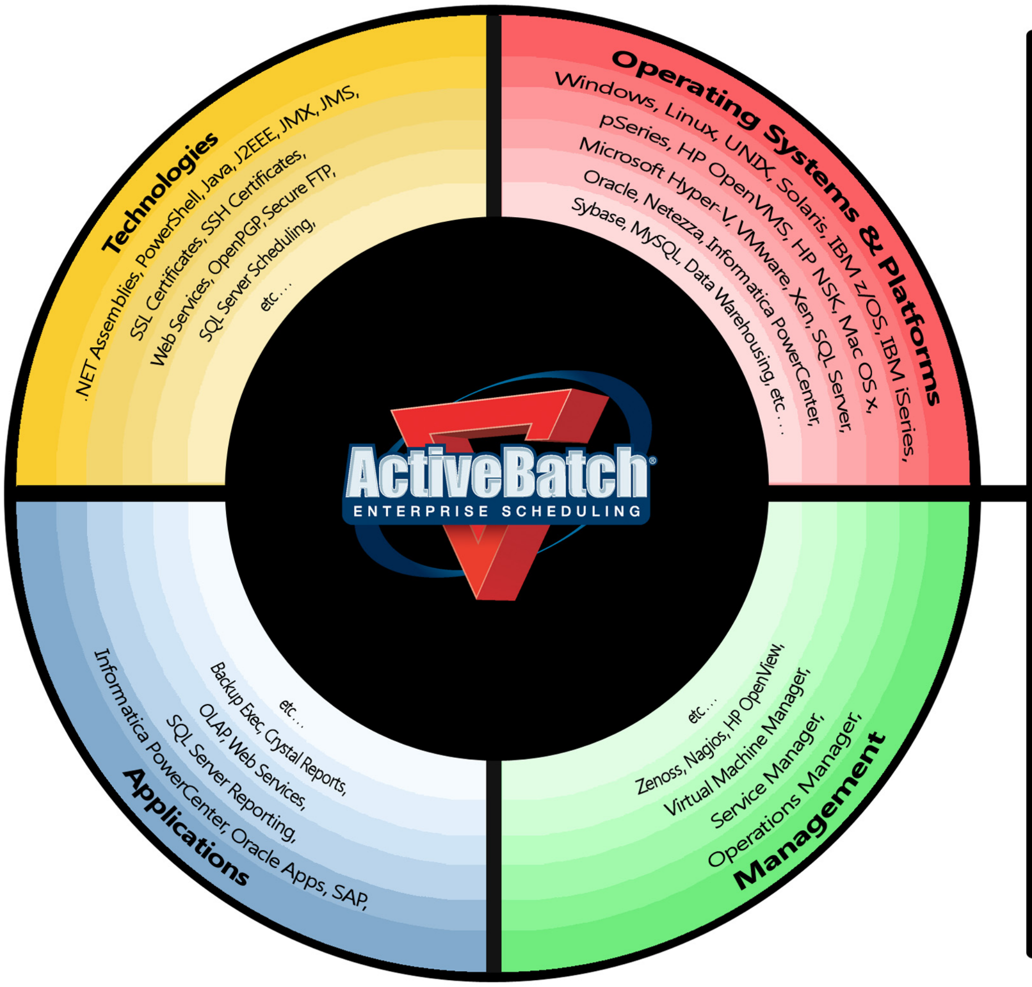
sponsored by

Advanced Systems Concepts, Inc.

Simplifying integration

Removing complexity

Shortening time to implementation


Technologies
.NET Assemblies, PowerShell, Java, J2EE, JMX, JMS,
SSL Certificates, SSH Certificates,
Web Services, OpenPGP, Secure FTP,
SQL Server Scheduling,
etc . . .
Operating Systems & Platforms
Windows, Linux, UNIX, Solaris, IBM z/OS, IBM iSeries,
pSeries, HP OpenVMS, HP NSK, Mac OS x,
Microsoft Hyper-V, VMware, Xen, SQL Server,
Oracle, Netezza, Informatica PowerCenter,
Sybase, MySQL, Data Warehousing, etc . . .
ActiveBatch
ENTERPRISE SCHEDULING
Applications
Informatica PowerCenter, Oracle Apps, SAP,
SQL Server Reporting,
OLAP Web Services,
Backup Exec, Crystal Reports,
etc . . .
Management
Operations Manager,
Service Manager,
Virtual Machine Manager,
Zenoss, Nagios, HP OpenView,
etc . . .

Business Benefits
Realized Through
ActiveBatch® Enterprise
Workload Automation

Integrate
Workflows Across Applications,
Platforms, Databases

Eliminate
Scheduling Wait Time

Improve
IT Service Levels
Server Utilization Rates

Implement
Centralized View and Management
of Jobs and Workflows

Increase
Effectiveness of Overall Business Processes

Reduce
Errors from Manual Operations
Cost of IT Operations

ROI:
139%

Payback Period:
4.6 Months

Integrating Your Business Applications Across Distributed Computing Environments

ActiveBatch® Enterprise Job Scheduling and Workload Automation, from Advanced Systems Concepts, removes the complexity from integrating business applications, databases, and mainframes. Focused on shortening the time to implementation and improving service levels by eliminating custom scripting via its Integrated Jobs Library, ActiveBatch automates job scheduling across diverse distributed computing environments, creating a centralized view of operations at the project, organizational or enterprise level.

Learn More at www.ActiveBatch.com

A|S C|I Advanced Systems Concepts, Inc.

Copyright Statement

Chapter 3: What Makes an IT Workflow? A Technical Deconstruction

Computers are useful because they'll perform an activity over and over without fail. The art is in telling them exactly what to do.

I hope that reading Chapter 2 was as enjoyable as writing it. Although I'll admit I took a little literary license in telling its stories, I did so to highlight the use cases where IT job scheduling makes perfect sense. Coordinating administrator activities, consolidating tasks, generalizing workflows, gathering data, orchestrating its transfer, triggering, and security are all important facets of regular data center administration. Yet too often these facets are administered using approaches that don't scale, introduce the potential for error, or can't be linked with other activities. The ultimate desire in each of Chapter 2's stories was the creation of *workflow*. That workflow absolutely involved each story's actors; but, more importantly, it involved the appropriate handling of those actors' data.

In many ways, workflows, jobs, and plans represent different facets of the same desire: *Telling a computer what to do*. You can consider them the logical representations of the "little automation packages" I referenced in the first two chapters. Although I spent much of those chapters explaining why they're good for your data center and how they'll benefit your distributed applications, I haven't yet shown you what they might look like.

That's what you'll see in this chapter. In it, you'll get an understanding of how a workflow quantifies an IT activity. You'll also walk through a set of mockups from a model IT job scheduling solution. Those mockups and the story that goes with them is intended to solidify your understanding of how an IT job scheduling solution might look once deployed.

But for now, let's stay at a high level for just a bit more. In doing so, I want to explain how workflows bring quantification to IT activities.

A Workflow Is an IT Activity, Quantified

A workflow has been described as *a sequence of connected steps*. More importantly, a workflow represents an abstraction of real work. It is a model that defines how data gets processed, where it goes, and what actions need to be accomplished with that data throughout its life cycle.

You can find workflows everywhere in business, and not all are technical in nature. Think about the last time you took a day off from work. You know that taking that day off requires first submitting a request. That request requires approval. Once approved, you notify teammates and coworkers of your impending unavailability. In the world of paid time off, you can't just miss a day without following that process.

And yet sometimes people do just miss days. Perhaps they were very sick, or got stuck on the side of the road far from cell phone service. In any of these cases, the workflow breaks down because the process isn't followed. What results is confusion about the person's whereabouts, and extra effort in figuring out what they were responsible for accomplishing during their absence.

You can compare this "people" workflow to the "data" workflows in an IT system. Data in an IT system needs to be handled appropriately. Actions on it must be scheduled with precision. Data must be transferred between systems in a timely manner. Failure states in processing need to be understood and handled. The result in any situation is a system where data and actions can be planned on.

To that end, let's explore further the IT plan first introduced back in Chapter 1. Figure 3.1 gives you a reproduction of the graphic you saw back in that chapter. There, you can see how three jobs have been gathered together to create *Plan 7 – Send Data Somewhere*.

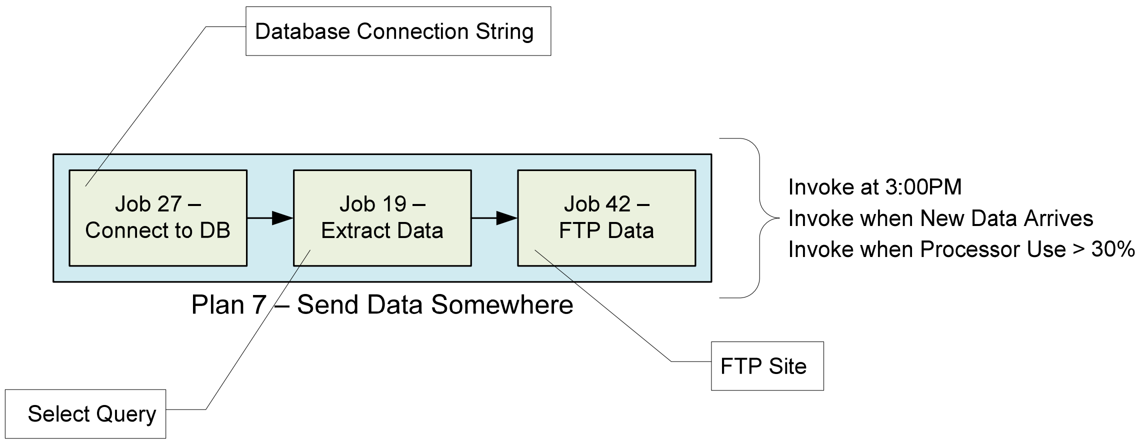

Database Connection String
Job 27 – Connect to DB
Job 19 – Extract Data
Job 42 – FTP Data
Invoke at 3:00PM
Invoke when New Data Arrives
Invoke when Processor Use > 30%
Plan 7 – Send Data Somewhere
FTP Site
Select Query

Figure 3.1: An example IT plan.

I won't explain again what this plan intends to do; the activities should be self-explanatory. More important is the recognition that this example shows how an IT workflow quantifies an activity along a set of axes: capturability, monitorability and measurability, repeatability and reusability, and finally security. Let's explore each.

Capturable

I find myself often repeating the statement, "Always remember, computers are deterministic!" Given the same input and processing instructions, they will always produce the same result. Yet even with this assertion, why do they sometimes *not* produce the result we're looking for?

That problem often centers on how well the established workflow captures the environment's potential states. A well-designed workflow (and the solution used to create it) must have the ability to capture a system's states and subsequently do something based on what it sees.

I recently heard a story that perfectly highlights this need for capturability. In that story, a company ran numerous mission-critical databases across more than one database platform. Most of these databases were part of homegrown applications that the company had created over time.

Backing up these databases was a regular chore for the IT department. Although the company's backup solution could indeed complete backups with little administrator input, the configuration of many databases required manual steps for backups to complete correctly. Due to simple human error, those manual steps sometimes weren't completed correctly. With more than 25 databases to manipulate, that human error became the biggest risk in the system. Fixing the problem was accomplished by implementing a solution that could capture the manual portions of the activity into an IT job.

Such capture is only possible when an IT job scheduling solution is richly instrumented. That solution must include the necessary vision into backup solutions, database solutions, and even custom codebases. Vision into every system component means *knowing when the task needs accomplishing*.

Monitorable and Measurable

You can't capture something unless you can monitor and measure it. Just as important as visibility into a system is visibility into the workflow surrounding that system. An effective IT job scheduling solution must be able to instrument its own activities so that the job itself can recover from any failure states.

This is of particular importance because most IT jobs don't operate interactively. Once created, tested, and set into production, a typical IT job is expected to accomplish its tasks without further assistance. This autonomy means that well-designed jobs must include monitoring and measurement components to know when data or actions are different from expected values.

It's easiest to understand this requirement by looking at the simple IT plan in Figure 3.1. Such a workflow is only useful when its activities are measurable. More important, measurement of a plan's logic must occur at multiple points throughout the plan's execution. Figure 3.2 shows how this built-in validation can be tagged to each phase of the plan's execution. In it, you see how the hand-off between Job 27 and Job 19 requires measuring the success of the first job. If Job 27 cannot successfully connect to the database, then continuing the plan will be unsuccessful at best *and damaging at worst*. You don't want bad data being eventually sent via FTP to a remote location.

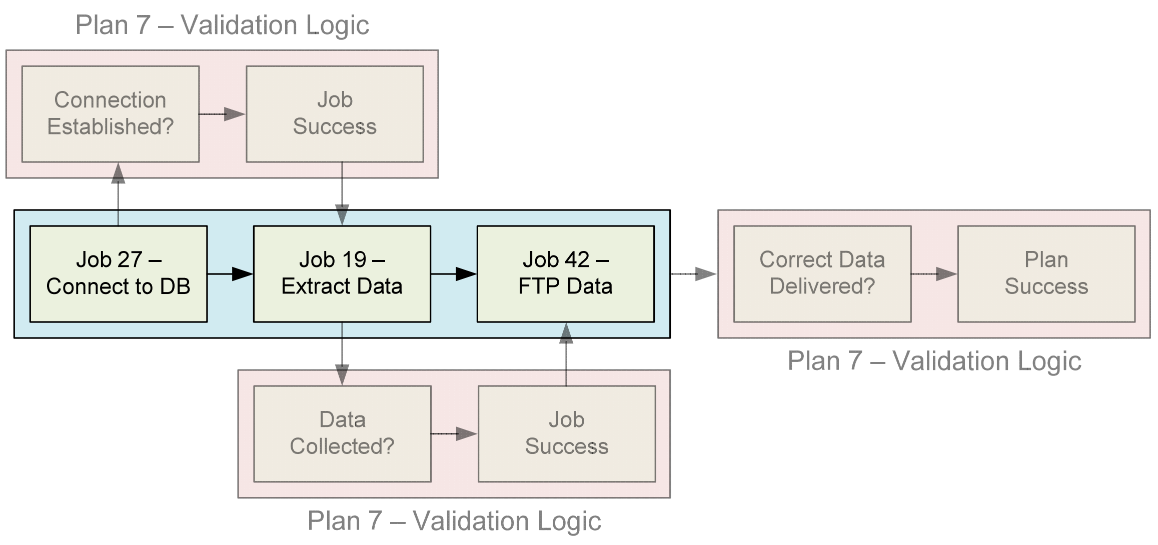


Figure 3.2: Validation logic ensures measurability.

Similar measurements must occur in the hand-off between Job 19 and Job 42 and again at plan completion. A successful IT job scheduling solution will create the workbench where validation logic like that shown in Figure 3.2 can be tagged throughout an IT plan. This logic should not impact the execution of individual jobs, nor is it necessarily part of whatever code runs beneath the job object. Effective solutions implement validation logic in such a way to be transparent to the execution of the job itself.

Repeatable and Reusable

Transparency of measurement along with parameterization of job objects combine to create a repeatable and reusable solution. You can imagine that creating a well-instrumented IT plan like Figure 3.2 is going to take some effort. Once expended, that effort gains extra value when it can be reused elsewhere.

Reusability comes into play not only within each IT job but also within each plan. Recall my assertion back in Chapter 1 that an IT job is "an action to be executed." This definition means that the boundary of an IT job must remain with the execution of an action. Figure 3.3 shows a graphical representation of an *Integrated Jobs Library*. In that library, is a collection of previously-created jobs: Job 17 updates a database row, Job 27 opens a connection to a database, and so on.

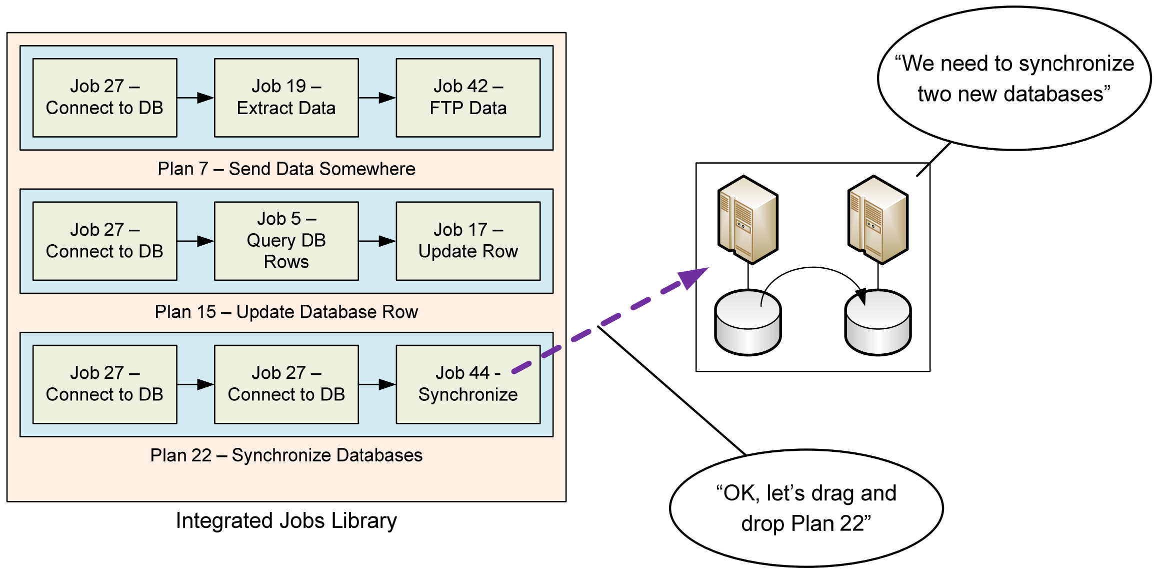


Figure 3.3: Reusing IT jobs in a plan; reusing IT plans in a workflow.

Each of those discrete jobs can be assigned to a workflow for the purposes of accomplishing some task. They can also be strung together in infinite combinations to create a more-powerful IT plan. You can see an example of this in Figure 3.3. Notice how Job 27 represents the beginning step of Plan 15; it also represents a middle step for Plan 22.

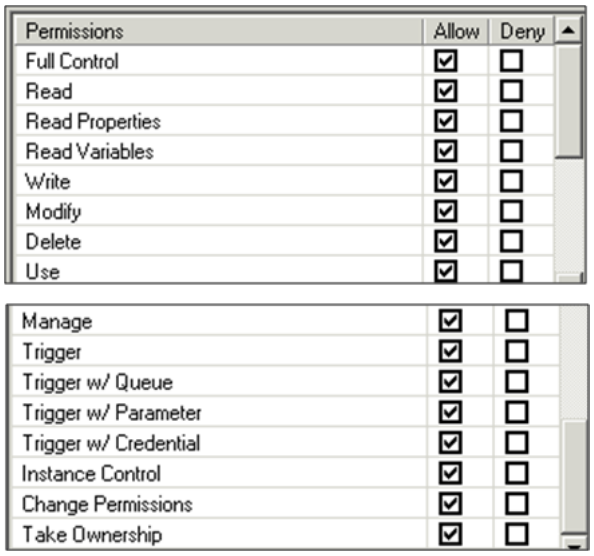
Once created, both jobs and plans reside in an IT job scheduling solution's Integrated Jobs Library. From there, created jobs can be reused repeatedly as similar tasks are required. In Figure 3.3, two new databases require synchronization. Since a plan has already been created to accomplish this task, reusing that plan elsewhere can be as simple as a drag-and-drop. After dragging to create a new instance of the plan, the only remaining activities involve populating that plan with new server characteristics.

Securable

Chapter 2 introduced the notion of job security. In the seventh story, you read how individual jobs and entire plans can be assigned security controls to prevent misuse. That level of security is indeed an important part of any IT system; however, Chapter 2 only began the conversation.

Consider the situation where an IT plan updates data in a database. Correctly constructing this IT plan requires parameterizing the plan to eliminate specific row values or items of data to update. However, parameterizing the plan in this way introduces the possibility that someone could accidentally (or maliciously) reassign the plan and update the wrong data.

This risk highlights why deep-level security is fundamentally important to an IT job scheduling solution. You want controls in place to protect someone from invoking a plan inappropriately. But you also want controls in place to protect certain instances of or triggers for that plan to be executed. Each platform and application tied into your IT job scheduling solution has its own security model, as does the job scheduling solution itself. Mapping these two layers together is what enables a job scheduling solution to, for example, apply Active Directory (AD) security principles to some application with a non-Windows' security model. Doing so enables you to lean on your existing AD infrastructure for the purpose of assigning rights and privileges in other platforms and applications. Figure 3.4 shows how such an extended access control list (ACL) might look, with triggers, trigger characteristics, and even instances of such a plan being individually securable.



Permissions	Allow	Deny
Full Control	☑	☐
Read	☑	☐
Read Properties	☑	☐
Read Variables	☑	☐
Write	☑	☐
Modify	☑	☐
Delete	☑	☐
Use	☑	☐
Manage	☑	☐
Trigger	☑	☐
Trigger w/ Queue	☑	☐
Trigger w/ Parameter	☑	☐
Trigger w/ Credential	☑	☐
Instance Control	☑	☐
Change Permissions	☑	☐
Take Ownership	☑	☐

Figure 3.4: Applying deep security to a job or plan.

A Ground-Up IT Workflow Construction

At its core, an IT workflow is still a piece of code. Some kinds of code a solution's vendor will create and include within a job scheduling solution. These represent the built-in job objects in your solution's Integrated Jobs Library. Other code must be custom-created by the administrators who use that solution. No vendor can create objects for every situation, so sometimes you'll be authoring your own. Notwithstanding who creates the code, at the end of the day, it is that code that needs to be scheduled for execution.

With this in mind, let's walk through an extended example of constructing a workflow out of individual parts. You can assume in this example that an IT job scheduling solution has been implemented and will be used to author the workflow.

A diagram of that workflow is shown in Figure 3.5. In it, each block represents an activity to be scheduled. Its story goes like this: Data in a system needs to be monitored for changes. As changes occur, an IT plan must be invoked to gather the changes, run scripts against the data, and move it around through file copy and FTP transfers. While all these processes occur, individual jobs within the workflow must trigger each other for execution as well as monitor for service availability.

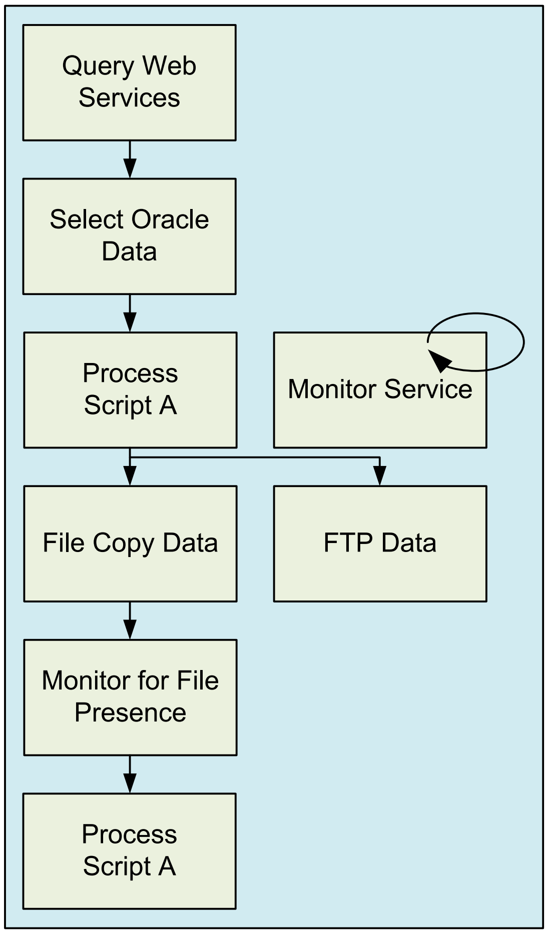
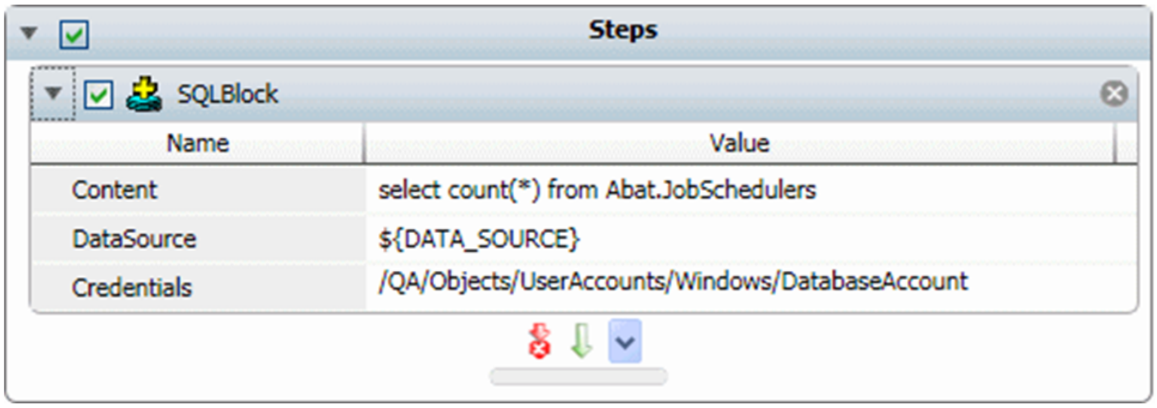


Figure 3.5: An example workflow.

You should immediately notice that scheduling is an important component of this workflow. That scheduling isn't accomplished through some clock-on-the-wall approach. It is instead based on monitoring the states present within the system (presence of files, WMI queries, log file changes, and so on), and firing subsequent actions based on changes in those states. *This intra-workflow triggering is the foundation of IT job scheduling.* Without it, scheduling jobs is little more than a function of time and date. A workflow like this requires a much faster response, one that moves from step to step based on the results of the just-completed step. You only get that through triggering.

Explaining Figure 3.5's workflow begins at its second step with the creation of an Oracle PL/SQL job object. This job object is necessary to run a query against the workflow's Oracle database. This object and its underlying query string should already be a component of your IT job scheduling solution. As a result, creating that job probably starts by clicking and dragging a representative SQL block (an example of which is shown in Figure 3.6) from a palette of options into the plan designer's workspace.

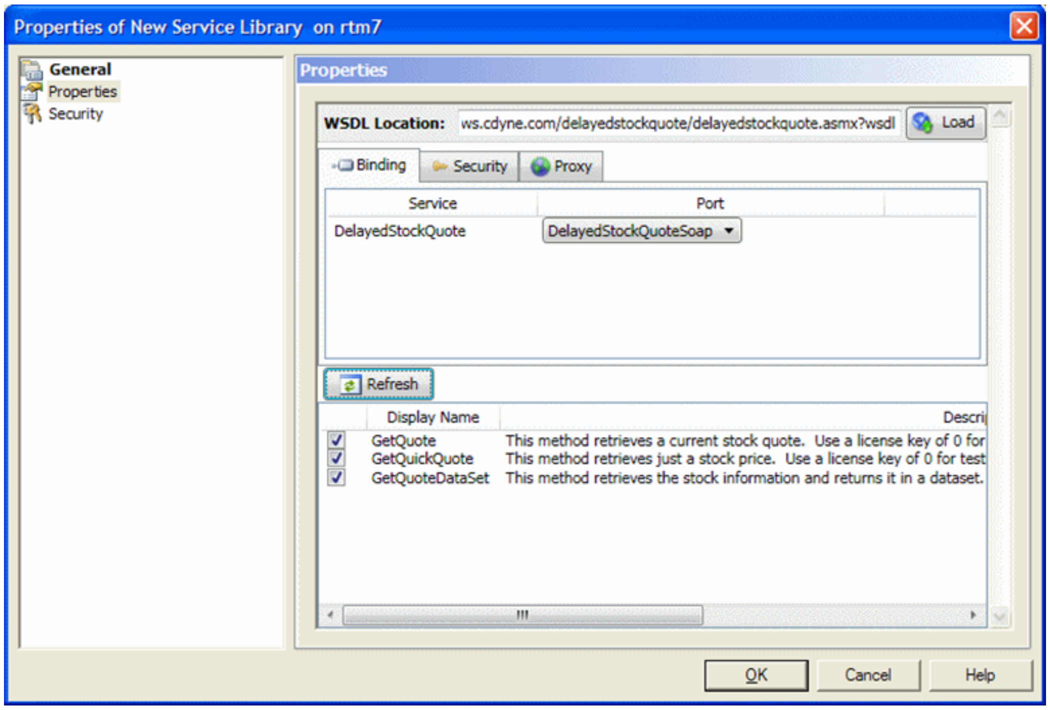


Name	Value
Content	select count(*) from Abat.JobSchedulers
DataSource	${DATA_SOURCE}
Credentials	/QA/Objects/UserAccounts/Windows/DatabaseAccount

Figure 3.6: Oracle PL/SQL object.

Once added to the workspace, specifics about this job object's use will then be added into the SQL block's properties screen. In Figure 3.6, you see how a SELECT statement is created to connect to an Oracle database and gather data. You should also notice how a variable—($DATA_SOURCE)—is used in this case to maintain the reusability of the job object.

Constructing that Oracle object is only the first step. By definition, there is no logic in it to define when it should be invoked. Accomplishing this requires creating one or more conditional statements. In this case, the workflow desires to query a Web service to see when data has changed. When it has, the Oracle SELECT statement is invoked. Figure 3.7 shows an example screen where such a Web services binding might be created. This binding identifies the methods that the Web service exposes, and is the first step in creating the necessary conditional logic.


Properties of New Service Library on rtm7

General
Properties
Security

Properties

WSDL Location: ws.cdyne.com/delayedstockquote/delayedstockquote.asmx?wsdl Load

Binding | Security | Proxy

Service	Port
DelayedStockQuote	DelayedStockQuoteSoap

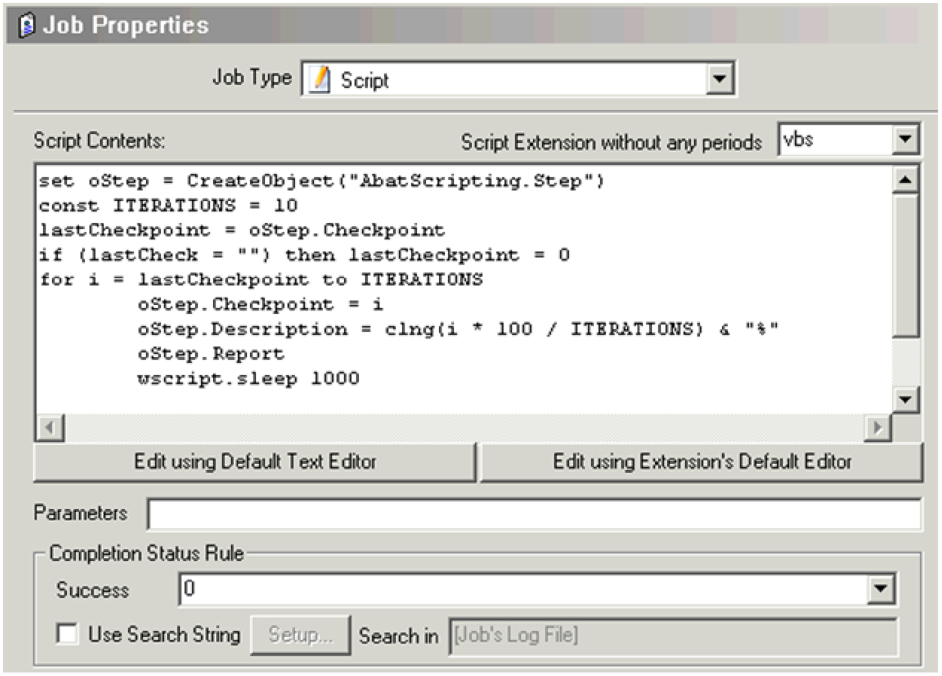
Refresh

	Display Name	Descri
✓	GetQuote	This method retrieves a current stock quote. Use a license key of 0 for
✓	GetQuickQuote	This method retrieves just a stock price. Use a license key of 0 for test
✓	GetQuoteDataSet	This method retrieves the stock information and returns it in a dataset.

OK Cancel Help

Figure 3.7: Web services connection.

Our example now includes conditional logic for monitoring the Web service for data changes. It also includes connection logic for gathering data from the Oracle database. The next step in the workflow requires processing that data through the use of a script block. Such a script block might be entered into an IT job scheduling solution using a wizard similar to Figure 3.8.


Job Properties

Job Type: Script

Script Contents: Script Extension without any periods: vbs

```
set oStep = CreateObject("AbatScripting.Step")
const ITERATIONS = 10
lastCheckpoint = oStep.Checkpoint
if (lastCheck = "") then lastCheckpoint = 0
for i = lastCheckpoint to ITERATIONS
        oStep.Checkpoint = i
        oStep.Description = clng(i * 100 / ITERATIONS) & "%"
        oStep.Report
        wscript.sleep 1000
```

Edit using Default Text Editor | Edit using Extension's Default Editor

Parameters:

Completion Status Rule
Success: 0
☐ Use Search String Setup... Search in [Job's Log File]

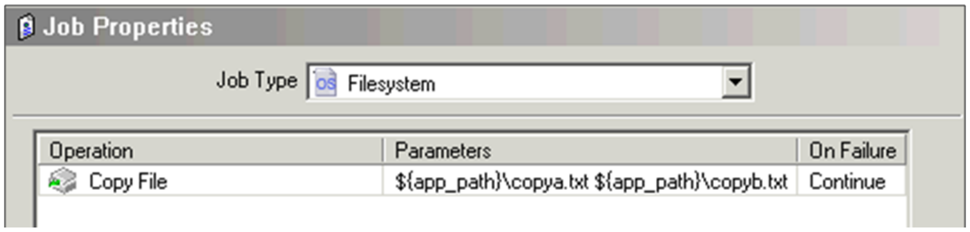
Figure 3.8: A scripting job.

In this mockup, a job object is created to bound a script. Scripting jobs are exceptionally malleable in that they can contain any code that is understood by the IT job scheduling solution and target application. In the case of Figure 3.9, the code is VBScript, although any supported code could be used.

The script's code is entered into the script block, along with other parameters like those seen in Figure 3.9: Those parameters are associated with the code itself, completion status, script extensions, and so on. Once created, the script becomes a job object just like the others in this workflow.

Note

As you can imagine, using custom code introduces the possibility for error into any IT plan. Your IT job scheduling solution will include scripting guidelines, but it should also include instrumentation to validate script variables and handle and alert on errors as they occur.



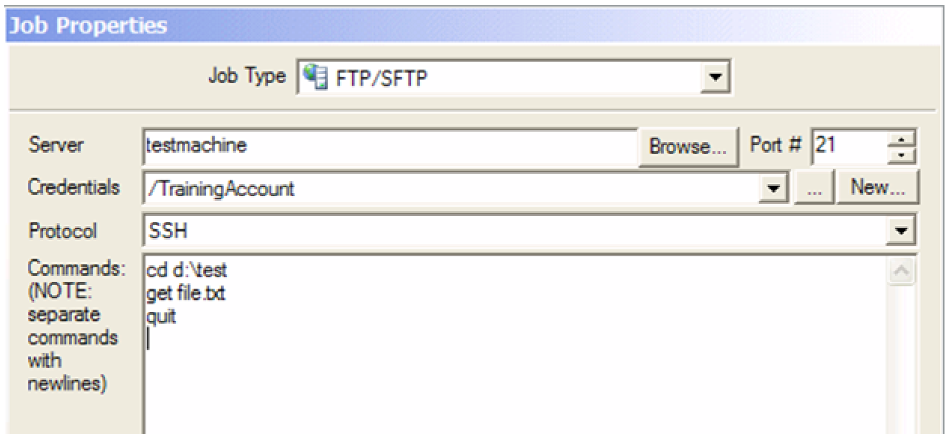
Operation	Parameters	On Failure
Copy File	${app_path}\copya.txt ${app_path}\copyb.txt	Continue

Figure 3.9: File copy job.

The next step in constructing the workflow is twofold. Figure 3.5's branching pattern illustrates the need to transfer the script's results to two locations using two different mechanisms. The first, seen in Figure 3.9, might be through a file copy job object.

Such an object is likely to be a built-in object within an IT job scheduling solution's Integrated Jobs Library. Thus, adding that job object to the plan may require little more than dragging it into the workspace just like with the SQL object. Once added, parameters associated with the file transfer are then added along with actions should a failure occur. Note again here how a variable is used in the file copy object's parameters to maintain reusability.

File copy jobs typically perform file transfers between similar operating systems (OSs), such as Microsoft Windows. But getting data off a Windows system and onto a Linux or UNIX system requires bridging protocols. That's why FTP jobs exist. Figure 3.10 shows how an FTP job object might look being dragged into the workspace. In Figure 3.10, an FTP (technically, an SFTP) job has been created. Added as parameters to that job are the FTP commands required to transfer the data as well as server names and credentials.



Job Type: FTP/SFTP
Server: testmachine
Port #: 21
Credentials: /TrainingAccount
Protocol: SSH
Commands: (NOTE: separate commands with newlines)

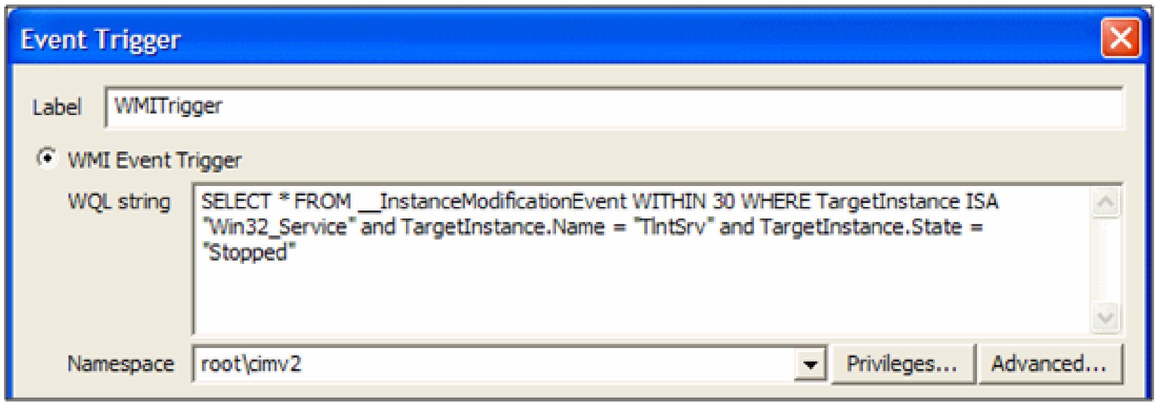
```
cd d:\test
get file.txt
quit
```

Figure 3.10: FTP job.

Note

Securing these credentials is also important to security. No regulated business or its auditors will look kindly on storing authentication credentials within an FTP command string. Thus, an effective IT job scheduling solution should provide a secured credentials store for such jobs. That store maintains credential security while allowing their reuse across multiple FTP jobs.

I mentioned earlier that monitoring and measurement were key components of good IT plan creation. If you're not monitoring your environment, you won't be prepared for unexpected states. One way to do that monitoring can be through a trigger. I show a portion of such a trigger in Figure 3.11.



Label: WMITrigger
(●) WMI Event Trigger
WQL string:

```
SELECT * FROM __InstanceModificationEvent WITHIN 30 WHERE TargetInstance ISA
"Win32_Service" and TargetInstance.Name = "TlntSrv" and TargetInstance.State =
"Stopped"
```

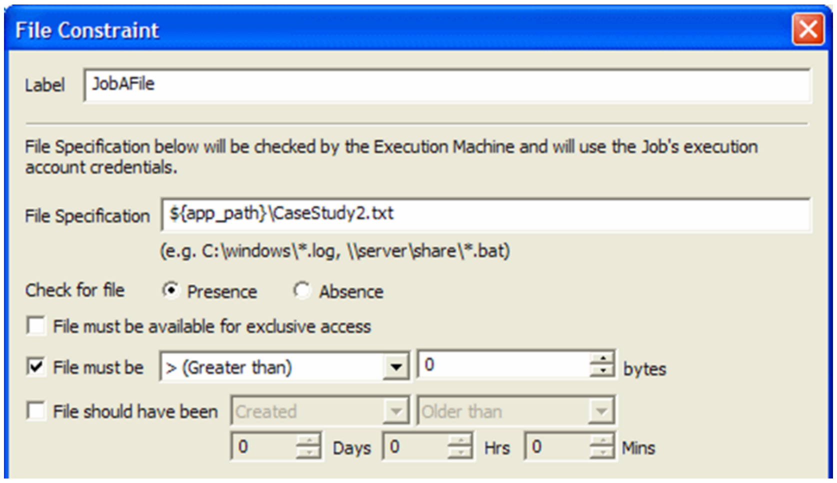
Namespace: root\cimv2

Figure 3.11: WMI-based trigger.

This trigger is used to facilitate the Monitor Service element in the workflow. For it, a Microsoft WMI query verifies the state of a service (in this case the TlntSrv or Telnet service). Not shown in the figure, but an important part of the job creation, is the action the trigger will accomplish when it discovers a stopped service. Assuming this sample workflow requires use of the service being monitored, the action associated with Figure 3.11 will be to restart that service if it is down.

This example is important because it highlights the kinds of state-correcting actions an IT job scheduling solution can automatically perform. If your workflow requires specific servers and their services (or daemons) to be operational, building those corrective measures directly into the workflow goes far into ensuring the continued operation of the distributed business system.

Our sample workflow needs to process two scripts to manipulate its data. The first you saw in Figure 3.8. I won't show you a similar view of the second script. Instead, I'll show you a constraint that might be applied (see Figure 3.12). Such a constraint can define when that script needs to be executed.


File Constraint

Label: JobAFile

File Specification below will be checked by the Execution Machine and will use the Job's execution account credentials.

File Specification: ${app_path}\CaseStudy2.txt

(e.g. C:\windows\*.log, \\server\share\*.bat)

Check for file: [✓ Presence] Absence

☐ File must be available for exclusive access

☑ File must be > (Greater than) 0 bytes

☐ File should have been Created Older than 0 Days 0 Hrs 0 Mins

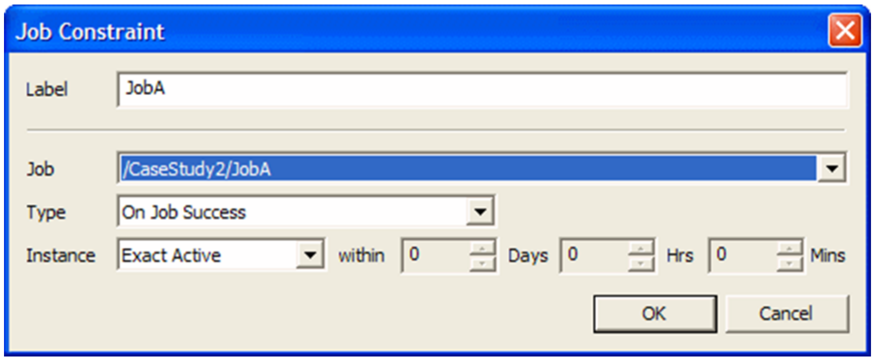
Figure 3.12: File constraint.

Recall that intra-workflow scheduling needs to be more than just time-based. Time-based schedulers are by nature insufficient because they can only process data at prescribed times of the day. Doing so creates inappropriate delay for workflow processing. What you really want is steps in a workflow to fire once a successful result from previous steps is verified.

You could achieve this by running the workflow line by line. However, doing so doesn't necessarily base the execution of following steps off results from previous steps. That's why Figure 3.12's file constraint is useful. Constraining an IT job's execution to occur only when a file is present allows that job to kick off only at the most appropriate time.

Our example workflow needs to process its second script after a file is copied. One can assume then that the copied file will be present on the target system. Thus, adding a file constraint to a job object means running the job only when the file is present and the previous step is complete.

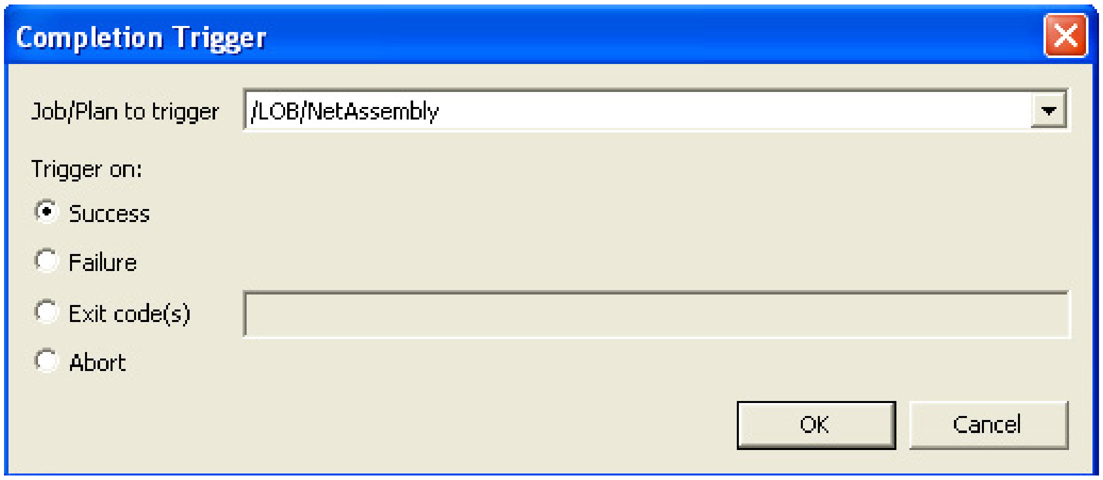
Although not necessarily related to this example, a pair of additional constraints is worth exploring. The first can be seen in Figure 3.13 where a job constraint has been placed on a job. For those plans where you simply want one job to follow another after its successful completion, job constraints can ensure that path is followed. Important to recognize here is that, as configured, whatever job follows the one in Figure 3.13 will only begin if the previous job is successful. Your IT job scheduling solution should include multiple options for defining when jobs in a plan are allowed to begin.



Label: JobA
Job: /CaseStudy2/JobA
Type: On Job Success
Instance: Exact Active within 0 Days 0 Hrs 0 Mins

Figure 3.13: Job constraint.

The other half of this equation is in telling which job to trigger after a successful completion. You can see an example of this in Figure 3.14. Here, a job (not identified in the figure) can be instructed to trigger upon the success of the previous job. Using combinations of constraints and triggers ensures that following steps in the workflow only execute when the state of the system is appropriate.

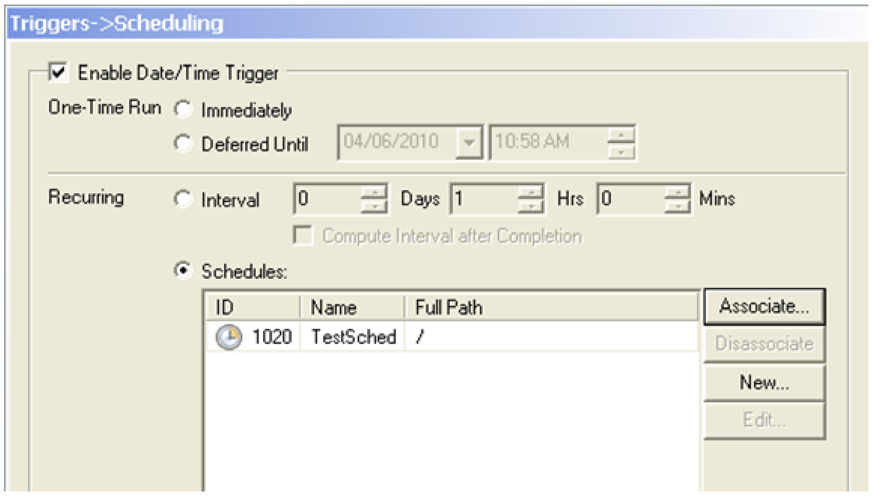


Job/Plan to trigger: /LOB/NetAssembly

Trigger on:
[✓ Success]
Failure
Exit code(s)
Abort

Figure 3.14: Completion trigger.

Although time-of-day scheduling is of comparatively minor use, it is still useful from time to time. Figure 3.15 shows an example scheduler that can be used for identifying when jobs should initiate. A good scheduler will include not only date- and time-based triggers but also scheduling support for complex scheduling needs.



ID	Name	Full Path
1020	TestSched	/

Figure 3.15: Time-based schedule.

Job Libraries and the Value of Triggers

Whatever IT job scheduling solution you choose needs to arrive with a suite of potential triggers that define when jobs are fired. These triggers perform multiple functions. They enable actions to be fired based on known states rather than requiring periodic "wake the script up and verify" batch jobs. They provide a mechanism to simplify event handling on external systems, a process that can be very complex when handled within a job object itself. They also create the potential for new types of actions, enacting change based on states that would otherwise be difficult to monitor within a script.

Consider the following possible triggers as a starting point for defining when you might want actions fired in your data center. This list gets you going. I'll expand on it in the next chapter, where I deliver a shopping list of capabilities you should look for in a solution:

- WMI Event Specifications (using WQL syntax)
- File event (across multiple platforms)
- Email event (across multiple email systems)
- Microsoft Message Queue event
- Web Services event
- Startup event
- SQL, Oracle, and other database event
- Virtual environment event

Last, although the core of any IT job scheduling solution is indeed the code that enacts changes on systems, the last thing you want to do is begin creating scripts if pre-created objects are already available. This chapter has discussed how an Integrated Jobs Library creates a palette of potential actions that you can add to your workspace. Figure 3.16 shows a representative sample of what one might look like. Pay careful attention to the actions that are available right out of the box in your chosen solution. You may find that leaning on your vendor for creating, testing, and validating objects greatly reduces your effort and risk of failure.

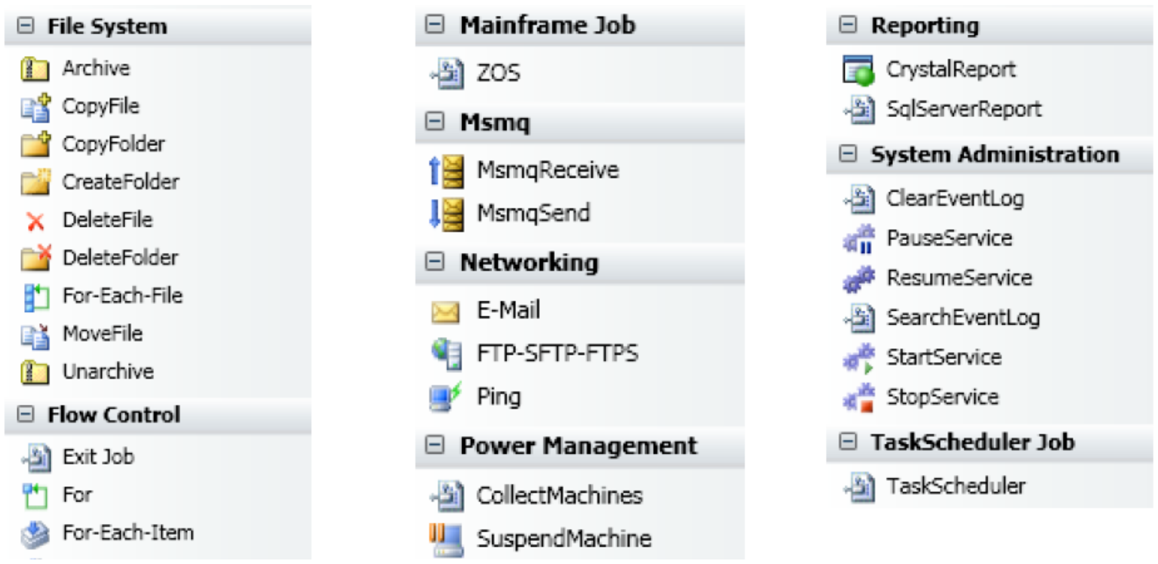


Figure 3.16: Integrated Job Library.

An IT Workflow "Tells the Computer What to Do"

Telling computers what to do is indeed an art, one that's bounded in the science of logic. Purchasing and implementing an IT job scheduling solution only nets you an empty palette within which you can create your own automations. That empty palette does, however, come with substantial capabilities for creating those instructions. This chapter has attempted to show you ways in which that might occur.

There's still one more story left to tell. That story deals with highlighting the capabilities that you might want in setting up that palette. That's the topic for the final chapter. In it, I'll share a shopping list of capabilities that you might look for in an IT job scheduling solution. Some of those features will probably make sense, while others might surprise you.