

Table of Contents

Overview

- [What is Azure Batch](#)

- [Feature overview](#)

- [APIs and tools](#)

- [Quotas and limits](#)

Get Started

- [Create a Batch account](#)

- [Batch .NET tutorial](#)

- [Batch Python tutorial](#)

How To

Develop

- [Application packages](#)

- [Autoscale compute nodes](#)

- [Azure AD authentication](#)

- [Concurrent node tasks](#)

- [Efficient list queries](#)

- [Job preparation and completion tasks](#)

- [Linux compute nodes](#)

- [Manage Batch accounts with Batch Management .NET](#)

- [Persist job and task output](#)

- [Run MPI jobs in Batch](#)

- [Task dependencies](#)

- [Visual Studio project templates for Batch](#)

Manage

- [Batch PowerShell cmdlets](#)

- [Azure CLI](#)

- [Batch diagnostic logs](#)

Reference

- [PowerShell](#)

[Azure CLI](#)

[.NET](#)

[Java](#)

[Node.js](#)

[Python SDK](#)

[REST](#)

[Related](#)

[Batch Shipyard](#)

[Batch and HPC solutions in the Azure cloud](#)

[Big Compute in Azure: technical resources](#)

[Resources](#)

[Pricing](#)

[MSDN forum](#)

[Stack Overflow](#)

[Videos](#)

[Service updates](#)

[C# code samples](#)

[Python code samples](#)

[Blog](#)

Run intrinsically parallel workloads with Batch

3/15/2017 • 5 min to read • [Edit Online](#)

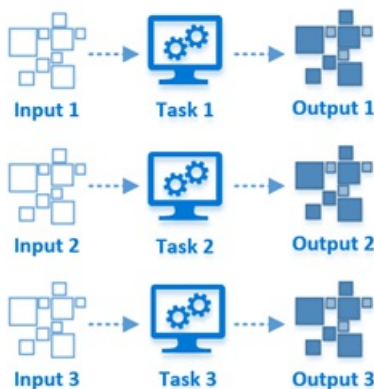
Azure Batch is a platform service for running large-scale parallel and high-performance computing (HPC) applications efficiently in the cloud. Azure Batch schedules compute-intensive work to run on a managed collection of virtual machines, and can automatically scale compute resources to meet the needs of your jobs.

With Azure Batch, you can easily define Azure compute resources to execute your applications in parallel, and at scale. There's no need to manually create, configure, and manage an HPC cluster, individual virtual machines, virtual networks, or a complex job and task scheduling infrastructure. Azure Batch automates or simplifies these tasks for you.

Use cases for Batch

Batch is a managed Azure service that is used for *batch processing* or *batch computing*--running a large volume of similar tasks for a desired result. Batch computing is most commonly used by organizations that regularly process, transform, and analyze large volumes of data.

Batch works well with intrinsically parallel (also known as "embarrassingly parallel") applications and workloads. Intrinsically parallel workloads are those that are easily split into multiple tasks that perform work simultaneously on many computers.



Some examples of workloads that are commonly processed using this technique are:

- Financial risk modeling
- Climate and hydrology data analysis
- Image rendering, analysis, and processing
- Media encoding and transcoding
- Genetic sequence analysis
- Engineering stress analysis
- Software testing

Batch can also perform parallel calculations with a reduce step at the end, and execute more complex HPC workloads such as [Message Passing Interface \(MPI\)](#) applications.

For a comparison between Batch and other HPC solution options in Azure, see [Batch and HPC solutions](#).

Pricing

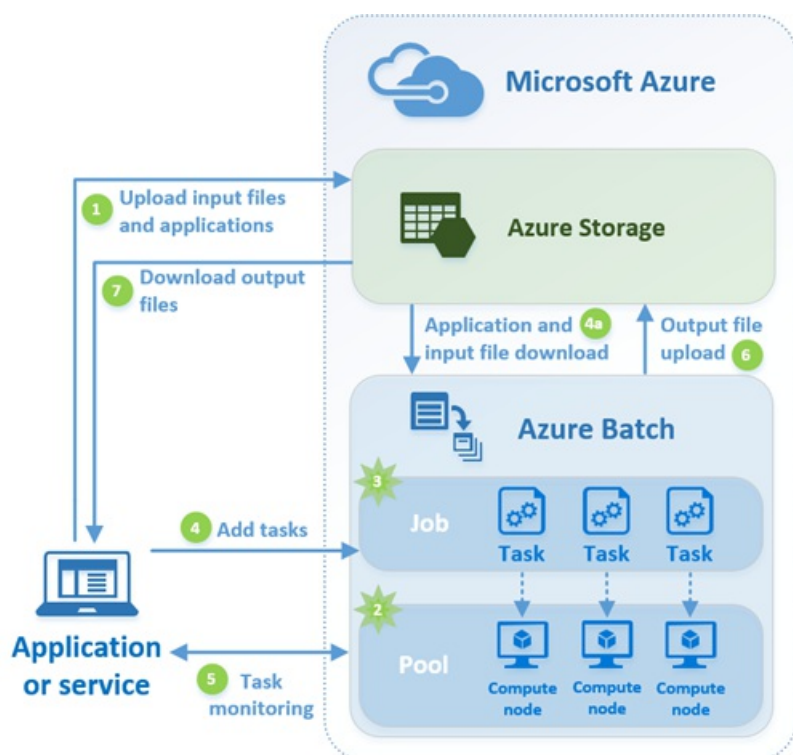
Azure Batch is a free service; you aren't charged for the Batch account itself. You are charged for the underlying

Azure compute resources that your Batch solutions consume, and for the resources consumed by other services when your workloads run. For example, you are charged for the compute nodes in your pools and for the data you store in Azure Storage as input or output for your tasks. Similarly, if you use the [application packages](#) feature of Batch, you are charged for the Azure Storage resources used for storing your application packages. See [Batch pricing](#) for more information.

Scenario: Scale out a parallel workload

A common solution that uses the Batch APIs to interact with the Batch service involves scaling out intrinsically parallel work--such as the rendering of images for 3D scenes--on a pool of compute nodes. This pool of compute nodes can be your "render farm" that provides tens, hundreds, or even thousands of cores to your rendering job, for example.

The following diagram shows a common Batch workflow, with a client application or hosted service using Batch to run a parallel workload.



In this common scenario, your application or service processes a computational workload in Azure Batch by performing the following steps:

1. Upload the **input files** and the **application** that will process those files to your Azure Storage account. The input files can be any data that your application will process, such as financial modeling data, or video files to be transcoded. The application files can be any application that is used for processing the data, such as a 3D rendering application or media transcoder.
2. Create a Batch **pool** of compute nodes in your Batch account--these nodes are the virtual machines that will execute your tasks. You specify properties such as the [node size](#), their operating system, and the location in Azure Storage of the application to install when the nodes join the pool (the application that you uploaded in step #1). You can also configure the pool to [automatically scale](#) in response to the workload that your tasks generate. Auto-scaling dynamically adjusts the number of compute nodes in the pool.
3. Create a Batch **job** to run the workload on the pool of compute nodes. When you create a job, you associate it with a Batch pool.
4. Add **tasks** to the job. When you add tasks to a job, the Batch service automatically schedules the tasks for execution on the compute nodes in the pool. Each task uses the application that you uploaded to process the input files.

- 4a. Before a task executes, it can download the data (the input files) that it is to process to the compute node it is assigned to. If the application has not already been installed on the node (see step #2), it can be downloaded here instead. When the downloads are complete, the tasks execute on their assigned nodes.
5. As the tasks run, you can query Batch to monitor the progress of the job and its tasks. Your client application or service communicates with the Batch service over HTTPS. Because you may be monitoring thousands of tasks running on thousands of compute nodes, be sure to [query the Batch service efficiently](#).
 6. As the tasks complete, they can upload their result data to Azure Storage. You can also retrieve files directly from the file system on a compute node.
 7. When your monitoring detects that the tasks in your job have completed, your client application or service can download the output data for further processing or evaluation.

Keep in mind this is just one way to use Batch, and this scenario describes only a few of its available features. For example, you can execute [multiple tasks in parallel](#) on each compute node, and you can use [job preparation and completion tasks](#) to prepare the nodes for your jobs, then clean up afterward.

Next steps

Now that you have a high-level overview of the Batch service, it's time to dig deeper to learn how you can use it to process your compute-intensive parallel workloads.

- Read the [Batch feature overview for developers](#), essential information for anyone preparing to use Batch. The article contains more detailed information about Batch service resources like pools, nodes, jobs, and tasks, and the many API features that you can use while building your Batch application.
- Learn about the [Batch APIs and tools](#) available for building Batch solutions.
- [Get started with the Azure Batch library for .NET](#) to learn how to use C# and the Batch .NET library to execute a simple workload using a common Batch workflow. This article should be one of your first stops while learning how to use the Batch service. There is also a [Python version](#) of the tutorial.
- Download the [code samples on GitHub](#) to see how both C# and Python can interface with Batch to schedule and process sample workloads.
- Check out the [Batch Learning Path](#) to get an idea of the resources available to you as you learn to work with Batch.

Develop large-scale parallel compute solutions with Batch

3/15/2017 • 35 min to read • [Edit Online](#)

In this overview of the core components of the Azure Batch service, we discuss the primary service features and resources that Batch developers can use to build large-scale parallel compute solutions.

Whether you're developing a distributed computational application or service that issues direct [REST API](#) calls or you're using one of the [Batch SDKs](#), you'll use many of the resources and features discussed in this article.

TIP

For a higher-level introduction to the Batch service, see [Basics of Azure Batch](#).

Batch service workflow

The following high-level workflow is typical of nearly all applications and services that use the Batch service for processing parallel workloads:

1. Upload the **data files** that you want to process to an [Azure Storage](#) account. Batch includes built-in support for accessing Azure Blob storage, and your tasks can download these files to [compute nodes](#) when the tasks are run.
2. Upload the **application files** that your tasks will run. These files can be binaries or scripts and their dependencies, and are executed by the tasks in your jobs. Your tasks can download these files from your Storage account, or you can use the [application packages](#) feature of Batch for application management and deployment.
3. Create a [pool](#) of compute nodes. When you create a pool, you specify the number of compute nodes for the pool, their size, and the operating system. When each task in your job runs, it's assigned to execute on one of the nodes in your pool.
4. Create a [job](#). A job manages a collection of tasks. You associate each job to a specific pool where that job's tasks will run.
5. Add [tasks](#) to the job. Each task runs the application or script that you uploaded to process the data files it downloads from your Storage account. As each task completes, it can upload its output to Azure Storage.
6. Monitor job progress and retrieve the task output from Azure Storage.

The following sections discuss these and the other resources of Batch that enable your distributed computational scenario.

NOTE

You need a [Batch account](#) to use the Batch service. Also, nearly all solutions use an [Azure Storage](#) account for file storage and retrieval. Batch currently supports only the **General purpose** storage account type, as described in step 5 of [Create a storage account](#) in [About Azure storage accounts](#).

Batch service resources

Some of the following resources--accounts, compute nodes, pools, jobs, and tasks--are required by all solutions that use the Batch service. Others, like job schedules and application packages, are helpful, but optional, features.

- [Account](#)
- [Compute node](#)
- [Pool](#)
- [Job](#)
 - [Job schedules](#)
- [Task](#)
 - [Start task](#)
 - [Job manager task](#)
 - [Job preparation and release tasks](#)
 - [Multi-instance task \(MPI\)](#)
 - [Task dependencies](#)
- [Application packages](#)

Account

A Batch account is a uniquely identified entity within the Batch service. All processing is associated with a Batch account. When you perform operations with the Batch service, you need both the account name and one of its account keys. You can [create an Azure Batch account using the Azure portal](#).

Compute node

A compute node is an Azure virtual machine (VM) that is dedicated to processing a portion of your application's workload. The size of a node determines the number of CPU cores, memory capacity, and local file system size that is allocated to the node. You can create pools of Windows or Linux nodes by using either Azure Cloud Services or Virtual Machines Marketplace images. See the following [Pool](#) section for more information on these options.

Nodes can run any executable or script that is supported by the operating system environment of the node. This includes *.exe, *.cmd, *.bat and PowerShell scripts for Windows--and binaries, shell, and Python scripts for Linux.

All compute nodes in Batch also include:

- A standard [folder structure](#) and associated [environment variables](#) that are available for reference by tasks.
- **Firewall** settings that are configured to control access.
- [Remote access](#) to both Windows (Remote Desktop Protocol (RDP)) and Linux (Secure Shell (SSH)) nodes.

Pool

A pool is a collection of nodes that your application runs on. The pool can be created manually by you, or automatically by the Batch service when you specify the work to be done. You can create and manage a pool that meets the resource requirements of your application. A pool can be used only by the Batch account in which it was created. A Batch account can have more than one pool.

Azure Batch pools build on top of the core Azure compute platform. They provide large-scale allocation, application installation, data distribution, health monitoring, and flexible adjustment of the number of compute nodes within a pool ([scaling](#)).

Every node that is added to a pool is assigned a unique name and IP address. When a node is removed from a pool, any changes that are made to the operating system or files are lost, and its name and IP address are released for future use. When a node leaves a pool, its lifetime is over.

When you create a pool, you can specify the following attributes:

- Compute node **operating system** and **version**

You have two options when you select an operating system for the nodes in your pool: **Virtual Machine Configuration** and **Cloud Services Configuration**.

Virtual Machine Configuration provides both Linux and Windows images for compute nodes from the [Azure Virtual Machines Marketplace](#). When you create a pool that contains Virtual Machine Configuration nodes, you must specify not only the size of the nodes, but also the **virtual machine image reference** and the Batch **node agent SKU** to be installed on the nodes. For more information about specifying these pool properties, see [Provision Linux compute nodes in Azure Batch pools](#).

Cloud Services Configuration provides Windows compute nodes *only*. Available operating systems for Cloud Services Configuration pools are listed in the [Azure Guest OS releases and SDK compatibility matrix](#). When you create a pool that contains Cloud Services nodes, you need to specify only the node size and its *OS Family*. When you create pools of Windows compute nodes, you most commonly use Cloud Services.

- The *OS Family* also determines which versions of .NET are installed with the OS.
- As with worker roles within Cloud Services, you can specify an *OS Version* (for more information on worker roles, see the [Tell me about cloud services](#) section in the [Cloud Services overview](#)).
- As with worker roles, we recommend that you specify `*` for the *OS Version* so that the nodes are automatically upgraded, and there is no work required to cater to newly released versions. The primary use case for selecting a specific OS version is to ensure application compatibility, which allows backward compatibility testing to be performed before allowing the version to be updated. After validation, the *OS Version* for the pool can be updated and the new OS image can be installed--any running tasks are interrupted and requeued.

- **Size of the nodes**

Cloud Services Configuration compute node sizes are listed in [Sizes for Cloud Services](#). Batch supports all Cloud Services sizes except `ExtraSmall`, `STANDARD_A1_V2`, and `STANDARD_A2_V2`.

Virtual Machine Configuration compute node sizes are listed in [Sizes for virtual machines in Azure](#) (Linux) and [Sizes for virtual machines in Azure](#) (Windows). Batch supports all Azure VM sizes except `STANDARD_A0` and those with premium storage (`STANDARD_GS`, `STANDARD_DS`, and `STANDARD_DSV2` series).

When selecting a compute node size, consider the characteristics and requirements of the applications you'll run on the nodes. Aspects like whether the application is multithreaded and how much memory it consumes can help determine the most suitable and cost-effective node size. It's typical to select a node size assuming one task will run on a node at a time. However, it is possible to have multiple tasks (and therefore multiple application instances) [run in parallel](#) on compute nodes during job execution. In this case, it is common to choose a larger node size to accommodate the increased demand of parallel task execution. See [Task scheduling policy](#) for more information.

All of the nodes in a pool are the same size. If you intend to run applications with differing system requirements and/or load levels, we recommend that you use separate pools.

- **Target number of nodes**

This is the number of compute nodes that you want to deploy in the pool. This is referred to as a *target* because, in some situations, your pool might not reach the desired number of nodes. A pool might not reach the desired number of nodes if it reaches the [core quota](#) for your Batch account--or if there is an auto-scaling formula that you have applied to the pool that limits the maximum number of nodes (see the following "Scaling policy" section).

- **Scaling policy**

For dynamic workloads, you can write and apply an [auto-scaling formula](#) to a pool. The Batch service periodically evaluates your formula and adjusts the number of nodes within the pool based on various

pool, job, and task parameters that you can specify.

- **Task scheduling policy**

The [max tasks per node](#) configuration option determines the maximum number of tasks that can be run in parallel on each compute node within the pool.

The default configuration specifies that one task at a time runs on a node, but there are scenarios where it is beneficial to have two or more tasks executed on a node simultaneously. See the [example scenario](#) in the [concurrent node tasks](#) article to see how you can benefit from multiple tasks per node.

You can also specify a *fill type* which determines whether Batch spreads the tasks evenly across all nodes in a pool, or packs each node with the maximum number of tasks before assigning tasks to another node.

- **Communication status** of compute nodes

In most scenarios, tasks operate independently and do not need to communicate with one another. However, there are some applications in which tasks must communicate, like [MPI scenarios](#).

You can configure a pool to allow **internode communication**, so that nodes within a pool can communicate at runtime. When internode communication is enabled, nodes in Cloud Services Configuration pools can communicate with each other on ports greater than 1100, and Virtual Machine Configuration pools do not restrict traffic on any port.

Note that enabling internode communication also impacts the placement of the nodes within clusters and might limit the maximum number of nodes in a pool because of deployment restrictions. If your application does not require communication between nodes, the Batch service can allocate a potentially large number of nodes to the pool from many different clusters and datacenters to enable increased parallel processing power.

- **Start task** for compute nodes

The optional *start task* executes on each node as that node joins the pool, and each time a node is restarted or reimaged. The start task is especially useful for preparing compute nodes for the execution of tasks, like installing the applications that your tasks run on the compute nodes.

- **Application packages**

You can specify [application packages](#) to deploy to the compute nodes in the pool. Application packages provide simplified deployment and versioning of the applications that your tasks run. Application packages that you specify for a pool are installed on every node that joins that pool, and every time a node is rebooted or reimaged. Application packages are currently unsupported on Linux compute nodes.

- **Network configuration**

You can specify the ID of an Azure [virtual network \(VNet\)](#) in which the pool's compute nodes should be created. See the [Pool network configuration](#) section for more information.

IMPORTANT

All Batch accounts have a default **quota** that limits the number of **cores** (and thus, compute nodes) in a Batch account. You can find the default quotas and instructions on how to [increase a quota](#) (such as the maximum number of cores in your Batch account) in [Quotas and limits for the Azure Batch service](#). If you find yourself asking "Why won't my pool reach more than X nodes?" this core quota might be the cause.

Job

A job is a collection of tasks. It manages how computation is performed by its tasks on the compute nodes in a

pool.

- The job specifies the **pool** in which the work is to be run. You can create a new pool for each job, or use one pool for many jobs. You can create a pool for each job that is associated with a job schedule, or for all jobs that are associated with a job schedule.
- You can specify an optional **job priority**. When a job is submitted with a higher priority than jobs that are currently in progress, the tasks for the higher-priority job are inserted into the queue ahead of tasks for the lower-priority jobs. Tasks in lower-priority jobs that are already running are not preempted.
- You can use job **constraints** to specify certain limits for your jobs:

You can set a **maximum wallclock time**, so that if a job runs for longer than the maximum wallclock time that is specified, the job and all of its tasks are terminated.

Batch can detect and then retry failed tasks. You can specify the **maximum number of task retries** as a constraint, including whether a task is *always* or *never* retried. Retrying a task means that the task is requested to be run again.

- Your client application can add tasks to a job, or you can specify a [job manager task](#). A job manager task contains the information that is necessary to create the required tasks for a job, with the job manager task being run on one of the compute nodes in the pool. The job manager task is handled specifically by Batch--it is queued as soon as the job is created, and is restarted if it fails. A job manager task is *required* for jobs that are created by a [job schedule](#) because it is the only way to define the tasks before the job is instantiated.
- By default, jobs remain in the active state when all tasks within the job are complete. You can change this behavior so that the job is automatically terminated when all tasks in the job are complete. Set the job's **onAllTasksComplete** property ([OnAllTasksComplete](#) in Batch .NET) to *terminatejob* to automatically terminate the job when all of its tasks are in the completed state.

Note that the Batch service considers a job with *no* tasks to have all of its tasks completed. Therefore, this option is most commonly used with a [job manager task](#). If you want to use automatic job termination without a job manager, you should initially set a new job's **onAllTasksComplete** property to *noaction*, then set it to *terminatejob* only after you've finished adding tasks to the job.

Job priority

You can assign a priority to jobs that you create in Batch. The Batch service uses the priority value of the job to determine the order of job scheduling within an account (this is not to be confused with a [scheduled job](#)). The priority values range from -1000 to 1000, with -1000 being the lowest priority and 1000 being the highest. To update the priority of a job, call the [Update the properties of a job](#) operation (Batch REST), or modify the [CloudJob.Priority](#) property (Batch .NET).

Within the same account, higher-priority jobs have scheduling precedence over lower-priority jobs. A job with a higher-priority value in one account does not have scheduling precedence over another job with a lower-priority value in a different account.

Job scheduling across pools is independent. Between different pools, it is not guaranteed that a higher-priority job is scheduled first if its associated pool is short of idle nodes. In the same pool, jobs with the same priority level have an equal chance of being scheduled.

Scheduled jobs

[Job schedules](#) enable you to create recurring jobs within the Batch service. A job schedule specifies when to run jobs and includes the specifications for the jobs to be run. You can specify the duration of the schedule--how long and when the schedule is in effect--and how frequently jobs are created during the scheduled period.

Task

A task is a unit of computation that is associated with a job. It runs on a node. Tasks are assigned to a node for

execution, or are queued until a node becomes free. Put simply, a task runs one or more programs or scripts on a compute node to perform the work you need done.

When you create a task, you can specify:

- The **command line** for the task. This is the command line that runs your application or script on the compute node.

It is important to note that the command line does not actually run under a shell. Therefore, it cannot natively take advantage of shell features like [environment variable](#) expansion (this includes the `PATH`). To take advantage of such features, you must invoke the shell in the command line--for example, by launching `cmd.exe` on Windows nodes or `/bin/sh` on Linux:

```
cmd /c MyTaskApplication.exe %MY_ENV_VAR%
```

```
/bin/sh -c MyTaskApplication $MY_ENV_VAR
```

If your tasks need to run an application or script that is not in the node's `PATH` or reference environment variables, invoke the shell explicitly in the task command line.

- **Resource files** that contain the data to be processed. These files are automatically copied to the node from Blob storage in a general-purpose Azure Storage account before the task's command line is executed. For more information, see the sections [Start task](#) and [Files and directories](#).
- The **environment variables** that are required by your application. For more information, see the [Environment settings for tasks](#) section.
- The **constraints** under which the task should execute. For example, constraints include the maximum time that the task is allowed to run, the maximum number of times a failed task should be retried, and the maximum time that files in the task's working directory are retained.
- **Application packages** to deploy to the compute node on which the task is scheduled to run. [Application packages](#) provide simplified deployment and versioning of the applications that your tasks run. Task-level application packages are especially useful in shared-pool environments, where different jobs are run on one pool, and the pool is not deleted when a job is completed. If your job has fewer tasks than nodes in the pool, task application packages can minimize data transfer since your application is deployed only to the nodes that run tasks.

In addition to tasks you define to perform computation on a node, the following special tasks are also provided by the Batch service:

- [Start task](#)
- [Job manager task](#)
- [Job preparation and release tasks](#)
- [Multi-instance tasks \(MPI\)](#)
- [Task dependencies](#)

Start task

By associating a **start task** with a pool, you can prepare the operating environment of its nodes. For example, you can perform actions like installing the applications that your tasks run or starting background processes. The start task runs every time a node starts, for as long as it remains in the pool--including when the node is first added to the pool and when it is restarted or reimaged.

A primary benefit of the start task is that it can contain all of the information that is necessary to configure a compute node and install the applications that are required for task execution. Therefore, increasing the number of nodes in a pool is as simple as specifying the new target node count. The start task provides the Batch service the information that is needed to configure the new nodes and get them ready for accepting tasks.

As with any Azure Batch task, you can specify a list of **resource files** in [Azure Storage](#), in addition to a **command**

line to be executed. The Batch service first copies the resource files to the node from Azure Storage, and then runs the command line. For a pool start task, the file list typically contains the task application and its dependencies.

However, the start task could also include reference data to be used by all tasks that are running on the compute node. For example, a start task's command line could perform a `robocopy` operation to copy application files (which were specified as resource files and downloaded to the node) from the start task's [working directory](#) to the [shared folder](#), and then run an MSI or `setup.exe`.

IMPORTANT

Batch currently supports *only* the **General purpose** storage account type, as described in step 5 of [Create a storage account](#) in [About Azure storage accounts](#). Your Batch tasks (including standard tasks, start tasks, job preparation tasks, and job release tasks) must specify resource files that reside *only* in **General purpose** storage accounts.

It is typically desirable for the Batch service to wait for the start task to complete before considering the node ready to be assigned tasks, but you can configure this.

If a start task fails on a compute node, then the state of the node is updated to reflect the failure, and the node is not assigned any tasks. A start task can fail if there is an issue copying its resource files from storage, or if the process executed by its command line returns a nonzero exit code.

If you add or update the start task for an *existing* pool, you must reboot its compute nodes for the start task to be applied to the nodes.

Job manager task

You typically use a **job manager task** to control and/or monitor job execution--for example, to create and submit the tasks for a job, determine additional tasks to run, and determine when work is complete. However, a job manager task is not restricted to these activities. It is a fully fledged task that can perform any actions that are required for the job. For example, a job manager task might download a file that is specified as a parameter, analyze the contents of that file, and submit additional tasks based on those contents.

A job manager task is started before all other tasks. It provides the following features:

- It is automatically submitted as a task by the Batch service when the job is created.
- It is scheduled to execute before the other tasks in a job.
- Its associated node is the last to be removed from a pool when the pool is being downsized.
- Its termination can be tied to the termination of all tasks in the job.
- A job manager task is given the highest priority when it needs to be restarted. If an idle node is not available, the Batch service might terminate one of the other running tasks in the pool to make room for the job manager task to run.
- A job manager task in one job does not have priority over the tasks of other jobs. Across jobs, only job-level priorities are observed.

Job preparation and release tasks

Batch provides job preparation tasks for pre-job execution setup. Job release tasks are for post-job maintenance or cleanup.

- **Job preparation task:** A job preparation task runs on all compute nodes that are scheduled to run tasks, before any of the other job tasks are executed. You can use a job preparation task to copy data that is shared by all tasks, but is unique to the job, for example.
- **Job release task:** When a job has completed, a job release task runs on each node in the pool that executed at least one task. You can use a job release task to delete data that is copied by the job preparation task, or to compress and upload diagnostic log data, for example.

Both job preparation and release tasks allow you to specify a command line to run when the task is invoked. They offer features like file download, elevated execution, custom environment variables, maximum execution duration, retry count, and file retention time.

For more information on job preparation and release tasks, see [Run job preparation and completion tasks on Azure Batch compute nodes](#).

Multi-instance task

A [multi-instance task](#) is a task that is configured to run on more than one compute node simultaneously. With multi-instance tasks, you can enable high-performance computing scenarios that require a group of compute nodes that are allocated together to process a single workload (like Message Passing Interface (MPI)).

For a detailed discussion on running MPI jobs in Batch by using the Batch .NET library, check out [Use multi-instance tasks to run Message Passing Interface \(MPI\) applications in Azure Batch](#).

Task dependencies

[Task dependencies](#), as the name implies, allow you to specify that a task depends on the completion of other tasks before its execution. This feature provides support for situations in which a "downstream" task consumes the output of an "upstream" task--or when an upstream task performs some initialization that is required by a downstream task. To use this feature, you must first enable task dependencies on your Batch job. Then, for each task that depends on another (or many others), you specify the tasks which that task depends on.

With task dependencies, you can configure scenarios like the following:

- *taskB* depends on *taskA* (*taskB* will not begin execution until *taskA* has completed).
- *taskC* depends on both *taskA* and *taskB*.
- *taskD* depends on a range of tasks, such as tasks *1* through *10*, before it executes.

Check out [Task dependencies in Azure Batch](#) and the [TaskDependencies](#) code sample in the [azure-batch-samples](#) GitHub repository for more in-depth details on this feature.

Environment settings for tasks

Each task executed by the Batch service has access to environment variables that it sets on compute nodes. This includes environment variables defined by the Batch service ([service-defined](#)) and custom environment variables that you can define for your tasks. The applications and scripts your tasks execute have access to these environment variables during execution.

You can set custom environment variables at the task or job level by populating the *environment settings* property for these entities. For example, see the [Add a task to a job](#) operation (Batch REST API), or the [CloudTask.EnvironmentSettings](#) and [CloudJob.CommonEnvironmentSettings](#) properties in Batch .NET.

Your client application or service can obtain a task's environment variables, both service-defined and custom, by using the [Get information about a task](#) operation (Batch REST) or by accessing the [CloudTask.EnvironmentSettings](#) property (Batch .NET). Processes executing on a compute node can access these and other environment variables on the node, for example, by using the familiar `%VARIABLE_NAME%` (Windows) or `$VARIABLE_NAME` (Linux) syntax.

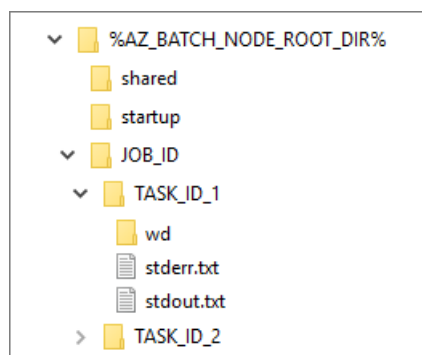
You can find a full list of all service-defined environment variables in [Compute node environment variables](#).

Files and directories

Each task has a *working directory* under which it creates zero or more files and directories. This working directory can be used for storing the program that is run by the task, the data that it processes, and the output of the processing it performs. All files and directories of a task are owned by the task user.

The Batch service exposes a portion of the file system on a node as the *root directory*. Tasks can access the root directory by referencing the `AZ_BATCH_NODE_ROOT_DIR` environment variable. For more information about using environment variables, see [Environment settings for tasks](#).

The root directory contains the following directory structure:



- **shared:** This directory provides read/write access to *all* tasks that run on a node. Any task that runs on the node can create, read, update, and delete files in this directory. Tasks can access this directory by referencing the `AZ_BATCH_NODE_SHARED_DIR` environment variable.
- **startup:** This directory is used by a start task as its working directory. All of the files that are downloaded to the node by the start task are stored here. The start task can create, read, update, and delete files under this directory. Tasks can access this directory by referencing the `AZ_BATCH_NODE_STARTUP_DIR` environment variable.
- **Tasks:** A directory is created for each task that runs on the node. It is accessed by referencing the `AZ_BATCH_TASK_DIR` environment variable.

Within each task directory, the Batch service creates a working directory (`wd`) whose unique path is specified by the `AZ_BATCH_TASK_WORKING_DIR` environment variable. This directory provides read/write access to the task. The task can create, read, update, and delete files under this directory. This directory is retained based on the *RetentionTime* constraint that is specified for the task.

`stdout.txt` and `stderr.txt` : These files are written to the task folder during the execution of the task.

IMPORTANT

When a node is removed from the pool, *all* of the files that are stored on the node are removed.

Application packages

The [application packages](#) feature provides easy management and deployment of applications to the compute nodes in your pools. You can upload and manage multiple versions of the applications run by your tasks, including their binaries and support files. Then you can automatically deploy one or more of these applications to the compute nodes in your pool.

You can specify application packages at the pool and task level. When you specify pool application packages, the application is deployed to every node in the pool. When you specify task application packages, the application is deployed only to nodes that are scheduled to run at least one of the job's tasks, just before the task's command line is run.

Batch handles the details of working with Azure Storage to store your application packages and deploy them to compute nodes, so both your code and management overhead can be simplified.

To find out more about the application package feature, check out [Application deployment with Azure Batch application packages](#).

NOTE

If you add pool application packages to an *existing* pool, you must reboot its compute nodes for the application packages to be deployed to the nodes.

Pool and compute node lifetime

When you design your Azure Batch solution, you have to make a design decision about how and when pools are created, and how long compute nodes within those pools are kept available.

On one end of the spectrum, you can create a pool for each job that you submit, and delete the pool as soon as its tasks finish execution. This maximizes utilization because the nodes are only allocated when needed, and shut down as soon as they're idle. While this means that the job must wait for the nodes to be allocated, it's important to note that tasks are scheduled for execution as soon as nodes are individually available, allocated, and the start task has completed. Batch does *not* wait until all nodes within a pool are available before assigning tasks to the nodes. This ensures maximum utilization of all available nodes.

At the other end of the spectrum, if having jobs start immediately is the highest priority, you can create a pool ahead of time and make its nodes available before jobs are submitted. In this scenario, tasks can start immediately, but nodes might sit idle while waiting for them to be assigned.

A combined approach is typically used for handling a variable, but ongoing, load. You can have a pool that multiple jobs are submitted to, but can scale the number of nodes up or down according to the job load (see [Scaling compute resources](#) in the following section). You can do this reactively, based on current load, or proactively, if load can be predicted.

Pool network configuration

When you create a pool of compute nodes in Azure Batch, you can specify the ID of an Azure [virtual network \(VNet\)](#) in which the pool's compute nodes should be created.

- Only **Cloud Services Configuration** pools can be assigned a VNet.
- The VNet must be:
 - In the same Azure **region** as the Azure Batch account.
 - In the same **subscription** as the Azure Batch account.
 - A **classic** VNet. VNets created with the Azure Resource Manager deployment model are not supported.
- The VNet should have enough free **IP addresses** to accommodate the `targetDedicated` property of the pool. If the subnet doesn't have enough free IP addresses, the Batch service partially allocates the compute nodes in the pool and returns a resize error.
- The *MicrosoftAzureBatch* service principal must have the [Classic Virtual Machine Contributor](#) Role-Based Access Control (RBAC) role for the specified VNet. In the Azure portal:
 - Select the **VNet**, then **Access control (IAM) > Roles > Classic Virtual Machine Contributor > Add**
 - Enter "MicrosoftAzureBatch" in the **Search** box
 - Check the **MicrosoftAzureBatch** check box
 - Select the **Select** button
- If communication to the compute nodes is denied by a **Network Security Group (NSG)** associated with the VNet, then the Batch service will set the state of the compute nodes to **unusable**. The subnet must allow communication from the Azure Batch service to be able to schedule tasks on the compute nodes.

Scaling compute resources

With [automatic scaling](#), you can have the Batch service dynamically adjust the number of compute nodes in a pool according to the current workload and resource usage of your compute scenario. This allows you to lower the overall cost of running your application by using only the resources you need, and releasing those you don't need.

You enable automatic scaling by writing an [automatic scaling formula](#) and associating that formula with a pool. The Batch service uses the formula to determine the target number of nodes in the pool for the next scaling interval (an interval that you can configure). You can specify the automatic scaling settings for a pool when you create it, or enable scaling on a pool later. You can also update the scaling settings on a scaling-enabled pool.

As an example, perhaps a job requires that you submit a very large number of tasks to be executed. You can assign a scaling formula to the pool that adjusts the number of nodes in the pool based on the current number of queued tasks and the completion rate of the tasks in the job. The Batch service periodically evaluates the formula and resizes the pool, based on workload and your other formula settings. The service adds nodes as needed when there are a large number of queued tasks, and removes nodes when there are no queued or running tasks.

A scaling formula can be based on the following metrics:

- **Time metrics** are based on statistics collected every five minutes in the specified number of hours.
- **Resource metrics** are based on CPU usage, bandwidth usage, memory usage, and number of nodes.
- **Task metrics** are based on task state, such as *Active* (queued), *Running*, or *Completed*.

When automatic scaling decreases the number of compute nodes in a pool, you must consider how to handle tasks that are running at the time of the decrease operation. To accommodate this, Batch provides a *node deallocation option* that you can include in your formulas. For example, you can specify that running tasks are stopped immediately, stopped immediately and then requeued for execution on another node, or allowed to finish before the node is removed from the pool.

For more information about automatically scaling an application, see [Automatically scale compute nodes in an Azure Batch pool](#).

TIP

To maximize compute resource utilization, set the target number of nodes to zero at the end of a job, but allow running tasks to finish.

Security with certificates

You typically need to use certificates when you encrypt or decrypt sensitive information for tasks, like the key for an [Azure Storage account](#). To support this, you can install certificates on nodes. Encrypted secrets are passed to tasks via command-line parameters or embedded in one of the task resources, and the installed certificates can be used to decrypt them.

You use the [Add certificate](#) operation (Batch REST) or [CertificateOperations.CreateCertificate](#) method (Batch .NET) to add a certificate to a Batch account. You can then associate the certificate with a new or existing pool. When a certificate is associated with a pool, the Batch service installs the certificate on each node in the pool. The Batch service installs the appropriate certificates when the node starts up, before launching any tasks (including the start task and job manager task).

If you add certificates to an *existing* pool, you must reboot its compute nodes for the certificates to be applied to the nodes.

Error handling

You might find it necessary to handle both task and application failures within your Batch solution.

Task failure handling

Task failures fall into these categories:

- **Scheduling failures**

If the transfer of files that are specified for a task fails for any reason, a *scheduling error* is set for the task.

Scheduling errors can occur if the task's resource files have moved, the Storage account is no longer available, or another issue was encountered that prevented the successful copying of files to the node.

- **Application failures**

The process that is specified by the task's command line can also fail. The process is deemed to have failed when a nonzero exit code is returned by the process that is executed by the task (see *Task exit codes* in the next section).

For application failures, you can configure Batch to automatically retry the task up to a specified number of times.

- **Constraint failures**

You can set a constraint that specifies the maximum execution duration for a job or task, the *maxWallClockTime*. This can be useful for terminating tasks that fail to progress.

When the maximum amount of time has been exceeded, the task is marked as *completed*, but the exit code is set to `0xC000013A` and the *schedulingError* field is marked as

```
{ category: "ServerError", code: "TaskEnded" } .
```

Debugging application failures

- `stderr` and `stdout`

During execution, an application might produce diagnostic output that you can use to troubleshoot issues. As mentioned in the earlier section [Files and directories](#), the Batch service writes standard output and standard error output to `stdout.txt` and `stderr.txt` files in the task directory on the compute node. You can use the Azure portal or one of the Batch SDKs to download these files. For example, you can retrieve these and other files for troubleshooting purposes by using [ComputeNode.GetNodeFile](#) and [CloudTask.GetNodeFile](#) in the Batch .NET library.

- **Task exit codes**

As mentioned earlier, a task is marked as failed by the Batch service if the process that is executed by the task returns a nonzero exit code. When a task executes a process, Batch populates the task's exit code property with the *return code of the process*. It is important to note that a task's exit code is **not** determined by the Batch service. A task's exit code is determined by the process itself or the operating system on which the process executed.

Accounting for task failures or interruptions

Tasks might occasionally fail or be interrupted. The task application itself might fail, the node on which the task is running might be rebooted, or the node might be removed from the pool during a resize operation if the pool's deallocation policy is set to remove nodes immediately without waiting for tasks to finish. In all cases, the task can be automatically requeued by Batch for execution on another node.

It is also possible for an intermittent issue to cause a task to hang or take too long to execute. You can set the maximum execution interval for a task. If the maximum execution interval is exceeded, the Batch service interrupts the task application.

Connecting to compute nodes

You can perform additional debugging and troubleshooting by signing in to a compute node remotely. You can

use the Azure portal to download a Remote Desktop Protocol (RDP) file for Windows nodes and obtain Secure Shell (SSH) connection information for Linux nodes. You can also do this by using the Batch APIs--for example, with [Batch .NET](#) or [Batch Python](#).

IMPORTANT

To connect to a node via RDP or SSH, you must first create a user on the node. To do this, you can use the Azure portal, [add a user account to a node](#) by using the Batch REST API, call the `ComputeNode.CreateComputeNodeUser` method in Batch .NET, or call the `add_user` method in the Batch Python module.

Troubleshooting problematic compute nodes

In situations where some of your tasks are failing, your Batch client application or service can examine the metadata of the failed tasks to identify a misbehaving node. Each node in a pool is given a unique ID, and the node on which a task runs is included in the task metadata. After you've identified a problem node, you can take several actions with it:

- **Reboot the node** ([REST](#) | [.NET](#))

Restarting the node can sometimes clear up latent issues like stuck or crashed processes. Note that if your pool uses a start task or your job uses a job preparation task, they are executed when the node restarts.

- **Reimage the node** ([REST](#) | [.NET](#))

This reinstalls the operating system on the node. As with rebooting a node, start tasks and job preparation tasks are rerun after the node has been reimaged.

- **Remove the node from the pool** ([REST](#) | [.NET](#))

Sometimes it is necessary to completely remove the node from the pool.

- **Disable task scheduling on the node** ([REST](#) | [.NET](#))

This effectively takes the node offline so that no further tasks are assigned to it, but allows the node to remain running and in the pool. This enables you to perform further investigation into the cause of the failures without losing the failed task's data, and without the node causing additional task failures. For example, you can disable task scheduling on the node, then [sign in remotely](#) to examine the node's event logs or perform other troubleshooting. After you've finished your investigation, you can then bring the node back online by enabling task scheduling ([REST](#) | [.NET](#)), or perform one of the other actions discussed earlier.

IMPORTANT

With each action that is described in this section--reboot, reimage, remove, and disable task scheduling--you are able to specify how tasks currently running on the node are handled when you perform the action. For example, when you disable task scheduling on a node by using the Batch .NET client library, you can specify a `DisableComputeNodeSchedulingOption` enum value to specify whether to **Terminate** running tasks, **Requeue** them for scheduling on other nodes, or allow running tasks to complete before performing the action (**TaskCompletion**).

Next steps

- Learn about the [Batch APIs and tools](#) available for building Batch solutions.
- Walk through a sample Batch application step-by-step in [Get started with the Azure Batch Library for .NET](#). There is also a [Python version](#) of the tutorial that runs a workload on Linux compute nodes.
- Download and build the [Batch Explorer](#) sample project for use while you develop your Batch solutions. Using the Batch Explorer, you can perform the following and more:

- Monitor and manipulate pools, jobs, and tasks within your Batch account
- Download `stdout.txt`, `stderr.txt`, and other files from nodes
- Create users on nodes and download RDP files for remote login
- Learn how to [create pools of Linux compute nodes](#).
- Visit the [Azure Batch forum](#) on MSDN. The forum is a good place to ask questions, whether you are just learning or are an expert in using Batch.

Overview of Batch APIs and tools

3/20/2017 • 4 min to read • [Edit Online](#)

Processing parallel workloads with Azure Batch is typically done programmatically by using one of the [Batch APIs](#). Your client application or service can use the Batch APIs to communicate with the Batch service. With the Batch APIs, you can create and manage pools of compute nodes, either virtual machines or cloud services. You can then schedule jobs and tasks to run on those nodes.

You can efficiently process large-scale workloads for your organization, or provide a service front end to your customers so that they can run jobs and tasks--on demand, or on a schedule--on one, hundreds, or even thousands of nodes. You can also use Azure Batch as part of a larger workflow, managed by tools such as [Azure Data Factory](#).

TIP

When you're ready to dig in to the Batch API for a more in-depth understanding of the features it provides, check out the [Batch feature overview for developers](#).

Azure accounts you'll need

When you develop Batch solutions, you'll use the following accounts in Microsoft Azure.

- **Azure account and subscription** - If you don't already have an Azure subscription, you can activate your [MSDN subscriber benefit](#), or sign up for a [free Azure account](#). When you create an account, a default subscription is created for you.
- **Batch account** - Azure Batch resources, including pools, compute nodes, jobs, and tasks, are associated with an Azure Batch account. When your application makes a request against the Batch service, it authenticates the request using the Azure Batch account name, the URL of the account, and an access key. You can [create Batch account](#) in the Azure portal.
- **Storage account** - Batch includes built-in support for working with files in [Azure Storage](#). Nearly every Batch scenario uses Azure Blob storage for staging the programs that your tasks run and the data that they process, and for the storage of output data that they generate. To create a Storage account, see [About Azure storage accounts](#).

Batch development APIs

Your applications and services can issue direct REST API calls or use one or more of the following client libraries to run and manage your Azure Batch workloads.

API	API REFERENCE	DOWNLOAD	TUTORIAL	CODE SAMPLES	MORE INFO
Batch REST	MSDN	N/A	-	-	Supported Versions
Batch .NET	docs.microsoft.com	NuGet	Tutorial	GitHub	Release Notes
Batch Python	readthedocs.io	PyPI	Tutorial	GitHub	Readme

API	API REFERENCE	DOWNLOAD	TUTORIAL	CODE SAMPLES	MORE INFO
Batch Node.js	github.io	npm	-	-	Readme
Batch Java (preview)	github.io	Maven	-	Readme	Readme

Batch command-line tools

Functionality provided by the development APIs is also available using command-line tools:

- [Batch PowerShell cmdlets](#): The Azure Batch cmdlets in the [Azure PowerShell](#) module enable you to manage Batch resources with PowerShell.
- [Azure CLI](#): The Azure Command-Line Interface (Azure CLI) is a cross-platform toolset that provides shell commands for interacting with many Azure services, including Batch.

Batch resource management

The Azure Resource Manager APIs for Batch provide programmatic access to Batch accounts. Using these APIs, you can programmatically manage Batch accounts, quotas, and application packages.

API	API REFERENCE	DOWNLOAD	TUTORIAL	CODE SAMPLES
Batch Resource Manager REST	docs.microsoft.com	N/A	-	GitHub
Batch Resource Manager .NET	docs.microsoft.com	NuGet	Tutorial	GitHub

Batch tools

While not required to build solutions using Batch, here are some valuable tools to use while building and debugging your Batch applications and services.

- [Azure portal](#): You can create, monitor, and delete Batch pools, jobs, and tasks in the Azure portal's Batch blades. You can view the status information for these and other resources while you run your jobs, and even download files from the compute nodes in your pools (download a failed task's `stderr.txt` while troubleshooting, for example). You can also download Remote Desktop (RDP) files that you can use to log in to compute nodes.
- [Azure Batch Explorer](#): Batch Explorer provides similar Batch resource management functionality as the Azure portal, but in a standalone Windows Presentation Foundation (WPF) client application. One of the Batch .NET sample applications available on [GitHub](#), you can build it with Visual Studio 2015 or newer and use it to browse and manage the resources in your Batch account while you develop and debug your Batch solutions. View job, pool, and task details, download files from compute nodes, and connect to nodes remotely by using Remote Desktop (RDP) files you can download with Batch Explorer.
- [Microsoft Azure Storage Explorer](#): While not strictly an Azure Batch tool, the Storage Explorer is another valuable tool to have while you are developing and debugging your Batch solutions.

Next steps

- Read the [Batch feature overview for developers](#), essential information for anyone preparing to use Batch. The article contains more detailed information about Batch service resources like pools, nodes, jobs, and tasks, and the many API features that you can use while building your Batch application.
- [Get started with the Azure Batch library for .NET](#) to learn how to use C# and the Batch .NET library to execute a

simple workload using a common Batch workflow. This article should be one of your first stops while learning how to use the Batch service. There is also a [Python version](#) of the tutorial.

- Download the [code samples on GitHub](#) to see how both C# and Python can interface with Batch to schedule and process sample workloads.
- Check out the [Batch Learning Path](#) to get an idea of the resources available to you as you learn to work with Batch.

Batch service quotas and limits

2/27/2017 • 2 min to read • [Edit Online](#)

As with other Azure services, there are limits on certain resources associated with the Batch service. Many of these limits are default quotas applied by Azure at the subscription or account level. This article discusses those defaults, and how you can request quota increases.

If you plan to run production workloads in Batch, you may need to increase one or more of the quotas above the default. If you want to raise a quota, you can open an online [customer support request](#) at no charge.

NOTE

A quota is a credit limit, not a capacity guarantee. If you have large-scale capacity needs, please contact Azure support.

Resource quotas

RESOURCE	DEFAULT LIMIT	MAXIMUM LIMIT
Batch accounts per region per subscription	3	50
Cores per Batch account	20	N/A ¹
Jobs and job schedules ² per Batch account	20	10,000
Pools per Batch account	20	5000

¹ The number of cores per Batch account can be increased, but the maximum number is unspecified. Contact customer support to discuss increase options.

² Includes run-once active jobs and active job schedules. Completed jobs and job schedules are not limited.

Other limits

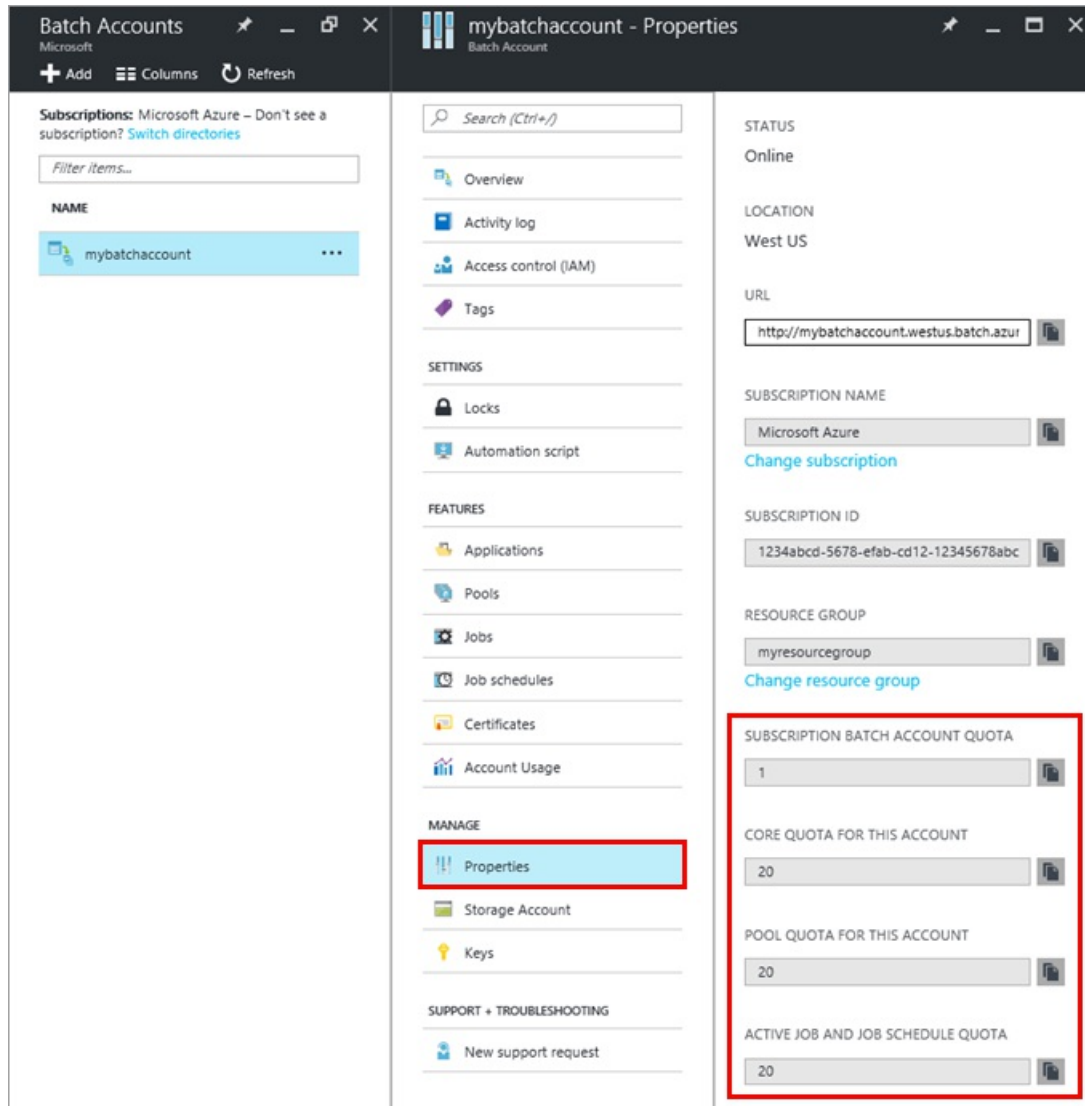
RESOURCE	MAXIMUM LIMIT
Concurrent tasks per compute node	4 x number of node cores
Applications per Batch account	20
Application packages per application	40
Application package size (each)	Approx. 195GB ¹

¹ Azure Storage limit for maximum block blob size

View Batch quotas

View your Batch account quotas in the [Azure portal](#).

1. Select **Batch accounts** in the portal, then select the Batch account you're interested in.
2. Select **Properties** on the Batch account's menu blade
3. The Properties blade displays the **quotas** currently applied to the Batch account



Increase a quota

Follow the steps below to request a quota increase using the [Azure portal](#).

1. Select the **Help + support** tile on your portal dashboard, or the question mark (?) in the upper-right corner of the portal.
2. Select **New support request > Basics**.
3. On the **Basics** blade:

- a. **Issue Type > Quota**
- b. Select your subscription.
- c. **Quota type > Batch**
- d. **Support plan > Quota support - Included**

Click **Next**.

4. On the **Problem** blade:

- a. Select a **Severity** according to your [business impact](#).

b. In **Details**, specify each quota you want to change, the Batch account name, and the new limit.

Click **Next**.

5. On the **Contact information** blade:

a. Select a **Preferred contact method**.

b. Verify and enter the required contact details.

Click **Create** to submit the support request.

Once you've submitted your support request, Azure support will contact you. Note that completing the request can take up to 2 business days.

Related topics

- [Create an Azure Batch account using the Azure portal](#)
- [Azure Batch feature overview](#)
- [Azure subscription and service limits, quotas, and constraints](#)

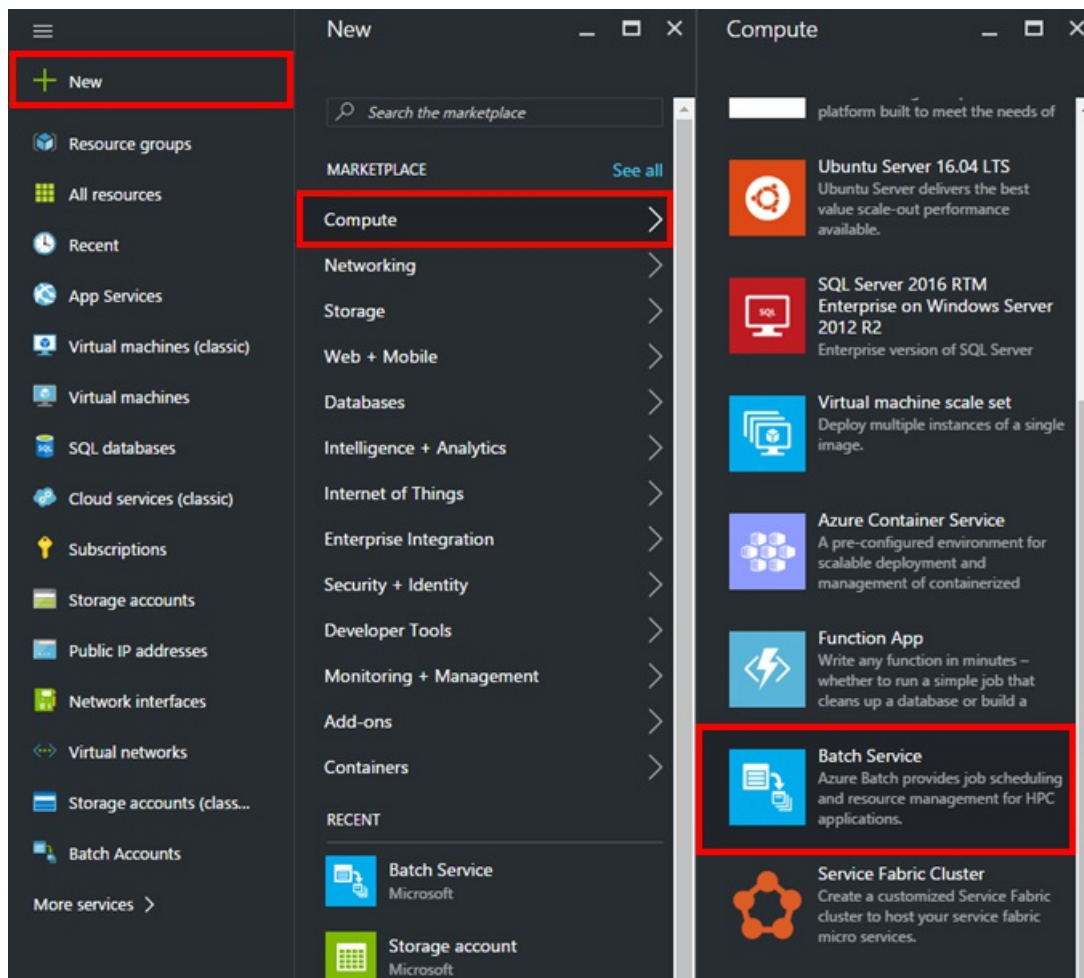
Create a Batch account with the Azure portal

3/20/2017 • 4 min to read • [Edit Online](#)

Learn how to create an Azure Batch account in the [Azure portal](#), and where to find important account properties like access keys and account URLs. We also discuss Batch pricing, and linking an Azure Storage account to your Batch account so that you can use [application packages](#) and [persist job and task output](#).

Create a Batch account

1. Sign in to the [Azure portal](#).
2. Click **New** > **Compute** > **Batch Service**.



3. The **New Batch Account** blade is displayed. See items *a* through *e* below for descriptions of each blade element.

The screenshot shows a web portal titled "New Batch Account" with the subtitle "Provide basic Batch account info". The form contains the following fields and options:

- Account name:** A text input field containing "mybatchaccount" with a green checkmark icon to its right. Below the field, the text ".eastus.batch.azure.com" is displayed.
- Subscription:** A dropdown menu showing "Visual Studio Enterprise with MSDN".
- Resource group:** Radio buttons for "Create new" and "Use existing" (which is selected). Below is a dropdown menu showing "myresourcegroup".
- Location:** A dropdown menu showing "East US".
- Storage Account:** A section with the text "Storage Account" and "Select a storage account" with a right-pointing chevron icon.
- Pin to dashboard:** A checkbox that is currently unchecked.
- Buttons:** A blue "Create" button and a link "Automation options".

a. **Account Name:** The name for your Batch account. The name you choose must be unique within the Azure region where the new account will be created (see **Location** below). The account name may contain only lowercase characters or numbers, and must be 3-24 characters in length.

b. **Subscription:** The subscription in which to create the Batch account. If you have only one subscription, it is selected by default.

c. **Resource group:** Select an existing resource group for your new Batch account, or optionally create a new one.

d. **Location:** The Azure region in which to create the Batch account. Only the regions supported by your subscription and resource group are displayed as options.

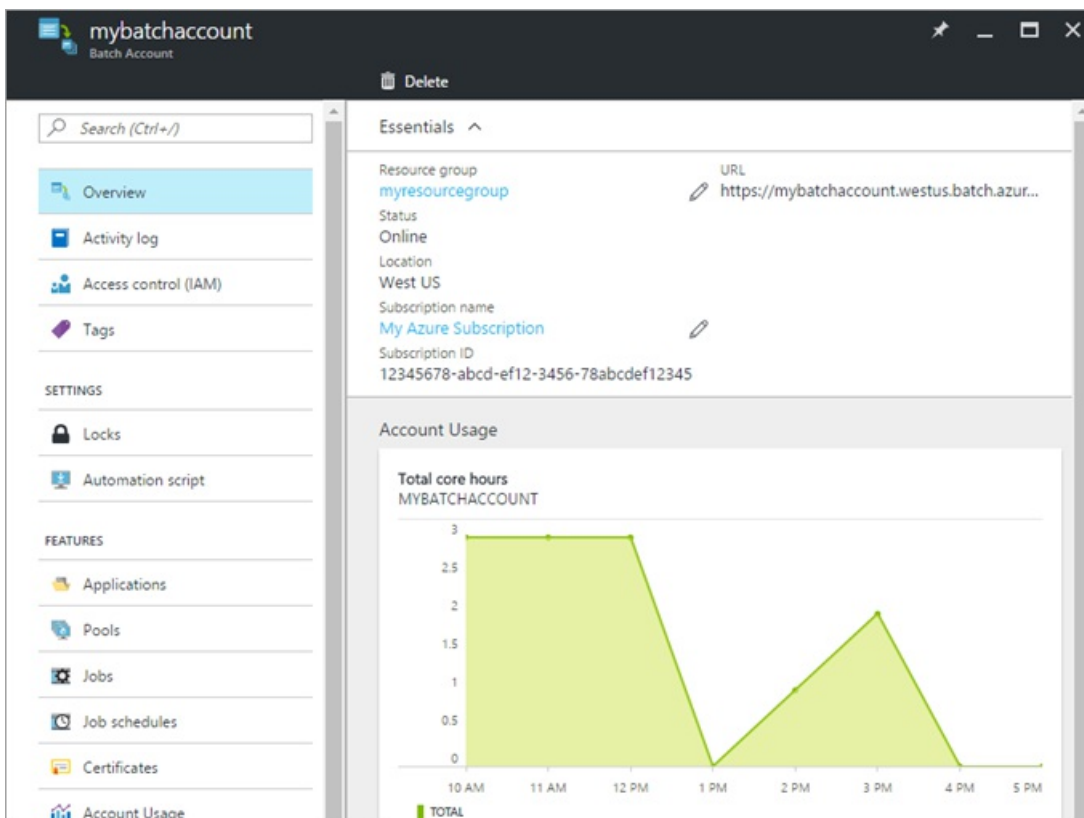
e. **Storage Account** (optional): A general-purpose Azure Storage account that you associate with your new Batch account. See [Linked Azure Storage account](#) below for more details.

4. Click **Create** to create the account.

The portal indicates that it is **Deploying** the account, and upon completion, a **Deployments succeeded** notification appears in *Notifications*.

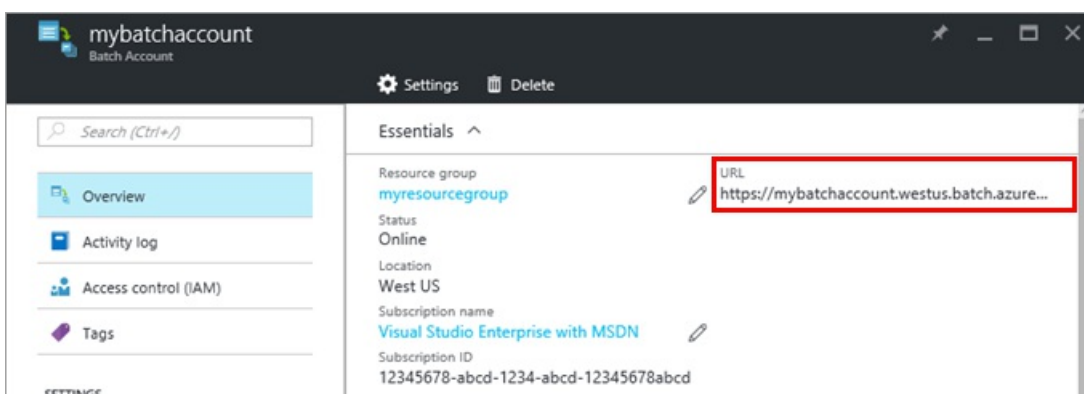
View Batch account properties

Once the account has been created, you can open the **Batch account blade** to access its settings and properties. You can access all account settings and properties by using the left menu of the Batch account blade.

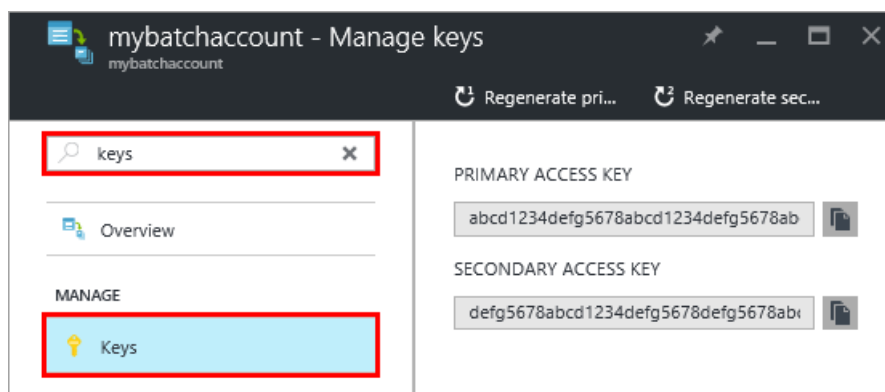


- **Batch account URL:** When you develop an application with the [Batch APIs](#), you'll need an account URL to access your Batch resources. A Batch account URL has the following format:

```
https://<account_name>.<region>.batch.azure.com
```



- **Access keys:** To authenticate access to your Batch account from your application, you'll need an account access key. To view or regenerate your Batch account's access keys, enter `keys` in the left menu **Search** box on the Batch account blade, then select **Keys**.



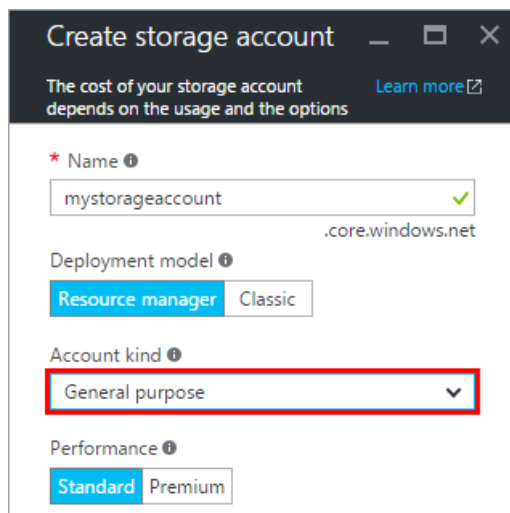
Pricing

Azure Batch is a free service; you aren't charged for the Batch account itself. You are charged for the underlying Azure compute resources that your Batch solutions consume, and for the resources consumed by other services when your workloads run. For example, you are charged for the compute nodes in your pools and for the data you store in Azure Storage as input or output for your tasks. Similarly, if you use the [application packages](#) feature of Batch, you are charged for the Azure Storage resources used for storing your application packages. See [Batch pricing](#) for more information.

Linked Azure Storage account

As mentioned earlier, you can optionally link a general-purpose Azure Storage account to your Batch account. The [application packages](#) feature of Batch uses Azure Blob storage, as does the [Batch File Conventions .NET](#) library. These optional features assist you in deploying the applications that your Batch tasks run, and persisting the data they produce.

We recommend that you create a new Storage account exclusively for use by your Batch account.



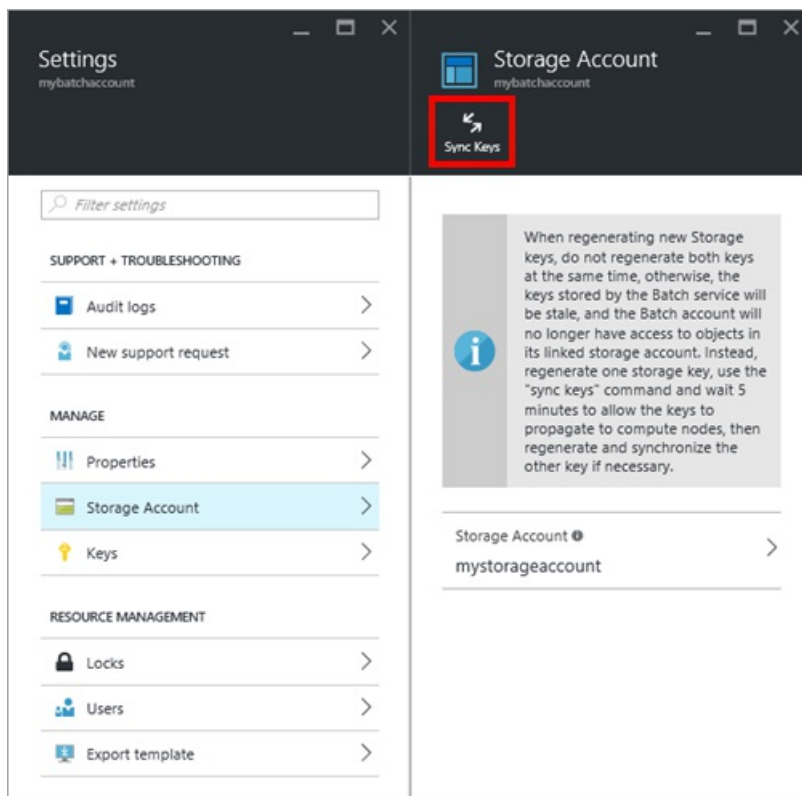
The screenshot shows the 'Create storage account' portal. At the top, it says 'Create storage account' with a close button. Below that, a note states 'The cost of your storage account depends on the usage and the options' with a 'Learn more' link. The form includes several fields: 'Name' with the value 'mystorageaccount' and a green checkmark, 'Deployment model' with 'Resource manager' selected, 'Account kind' with 'General purpose' selected (highlighted by a red rectangle), and 'Performance' with 'Standard' selected. The domain '.core.windows.net' is visible next to the name field.

NOTE

Azure Batch currently supports only the general-purpose Storage account type. This account type is described in step 5, [Create a storage account](#), in [About Azure storage accounts](#).

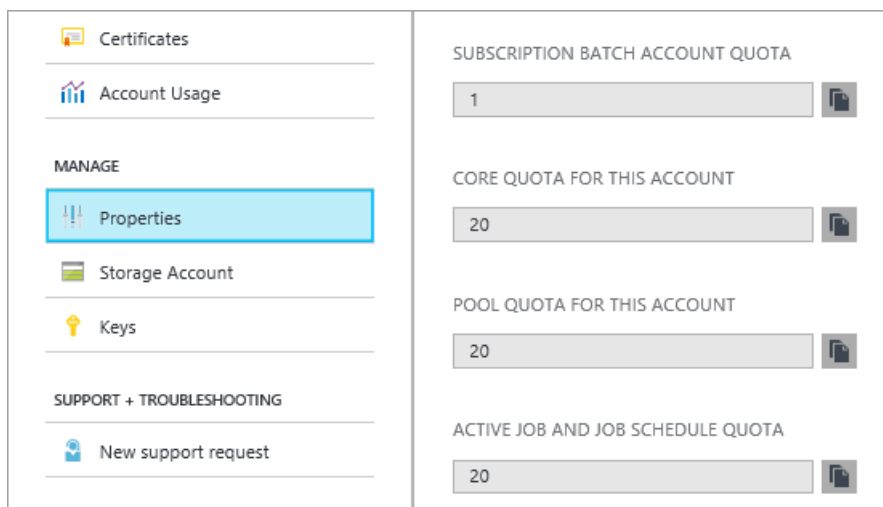
WARNING

Be careful when regenerating the access keys of a linked Storage account. Regenerate only one Storage account key and click **Sync Keys** on the linked Storage account blade. Wait five minutes to allow the keys to propagate to the compute nodes in your pools, then regenerate and synchronize the other key if necessary. If you regenerate both keys at the same time, your compute nodes will not be able to synchronize either key, and they will lose access to the Storage account.



Batch service quotas and limits

Please be aware that as with your Azure subscription and other Azure services, certain [quotas and limits](#) apply to Batch accounts. Current quotas for a Batch account appear in the portal in the account **Properties**.



Keep these quotas in mind as you are designing and scaling up your Batch workloads. For example, if your pool isn't reaching the target number of compute nodes you've specified, you might have reached the core quota limit for your Batch account.

The quota for Batch accounts is per region per subscription, so you can have more than one Batch account by default, as long as they are in different regions. You can run multiple Batch workloads in a single Batch account, or distribute your workloads among Batch accounts that are in the same subscription, but in different Azure regions.

Additionally, many of these quotas can be increased simply with a free product support request submitted in the Azure portal. See [Quotas and limits for the Azure Batch service](#) for details on requesting quota increases.

Other Batch account management options

In addition to using the Azure portal, you can also create and manage Batch accounts with the following:

- [Batch PowerShell cmdlets](#)
- [Azure CLI](#)
- [Batch Management .NET](#)

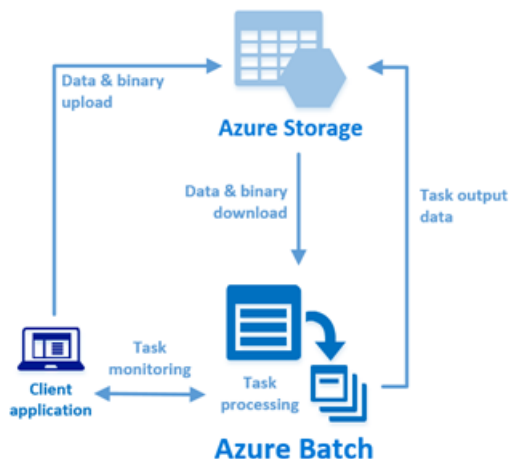
Next steps

- See the [Azure Batch feature overview](#) to learn more about Batch service concepts and features. The article discusses the primary Batch resources such as pools, compute nodes, jobs, and tasks, and provides an overview of the service's features that enable large-scale compute workload execution.
- Learn the basics of developing a Batch-enabled application using the [Batch .NET client library](#). The [introductory article](#) guides you through a working application that uses the Batch service to execute a workload on multiple compute nodes, and includes using Azure Storage for workload file staging and retrieval.

Get started building solutions with the Batch client library for .NET

3/28/2017 • 26 min to read • [Edit Online](#)

Learn the basics of [Azure Batch](#) and the [Batch .NET](#) library in this article as we discuss a C# sample application step by step. We look at how the sample application leverages the Batch service to process a parallel workload in the cloud, and how it interacts with [Azure Storage](#) for file staging and retrieval. You'll learn a common Batch application workflow and gain a base understanding of the major components of Batch such as jobs, tasks, pools, and compute nodes.



Prerequisites

This article assumes that you have a working knowledge of C# and Visual Studio. It also assumes that you're able to satisfy the account creation requirements that are specified below for Azure and the Batch and Storage services.

Accounts

- **Azure account:** If you don't already have an Azure subscription, [create a free Azure account](#).
- **Batch account:** Once you have an Azure subscription, [create an Azure Batch account](#).
- **Storage account:** See [Create a storage account](#) in [About Azure storage accounts](#).

IMPORTANT

Batch currently supports *only* the **General purpose** storage account type, as described in step #5 [Create a storage account](#) in [About Azure storage accounts](#).

Visual Studio

You must have **Visual Studio 2015 or newer** to build the sample project. You can find free and trial versions of Visual Studio in the [overview of Visual Studio products](#).

DotNetTutorial code sample

The [DotNetTutorial](#) sample is one of the many Batch code samples found in the [azure-batch-samples](#) repository on GitHub. You can download all the samples by clicking **Clone or download > Download ZIP** on the repository home page, or by clicking the [azure-batch-samples-master.zip](#) direct download link. Once you've extracted the contents of the ZIP file, you can find the solution in the following folder:

Azure Batch Explorer (optional)

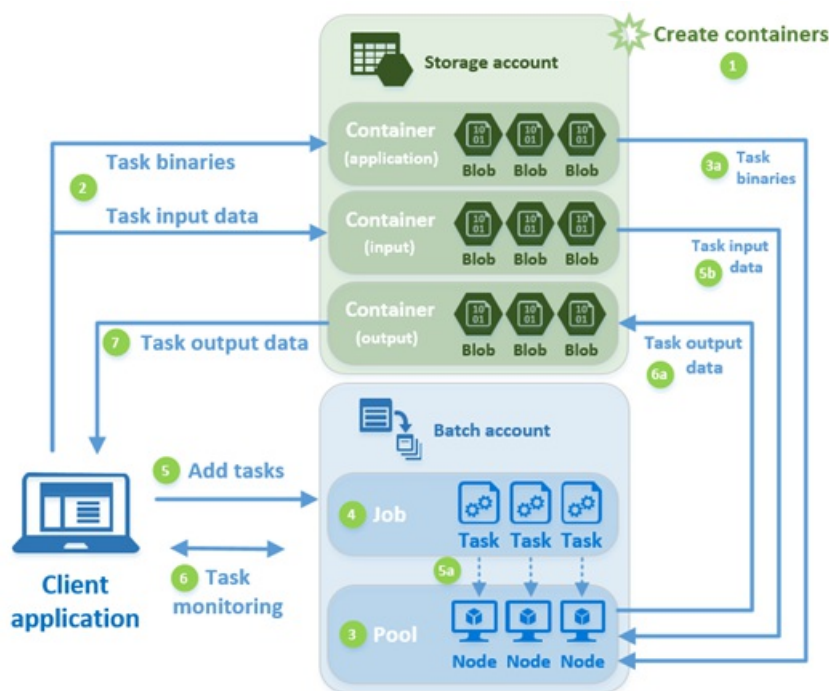
The [Azure Batch Explorer](#) is a free utility that is included in the [azure-batch-samples](#) repository on GitHub. While not required to complete this tutorial, it can be useful while developing and debugging your Batch solutions.

DotNetTutorial sample project overview

The *DotNetTutorial* code sample is a Visual Studio solution that consists of two projects: **DotNetTutorial** and **TaskApplication**.

- **DotNetTutorial** is the client application that interacts with the Batch and Storage services to execute a parallel workload on compute nodes (virtual machines). DotNetTutorial runs on your local workstation.
- **TaskApplication** is the program that runs on compute nodes in Azure to perform the actual work. In the sample, `TaskApplication.exe` parses the text in a file downloaded from Azure Storage (the input file). Then it produces a text file (the output file) that contains a list of the top three words that appear in the input file. After it creates the output file, TaskApplication uploads the file to Azure Storage. This makes it available to the client application for download. TaskApplication runs in parallel on multiple compute nodes in the Batch service.

The following diagram illustrates the primary operations that are performed by the client application, *DotNetTutorial*, and the application that is executed by the tasks, *TaskApplication*. This basic workflow is typical of many compute solutions that are created with Batch. While it does not demonstrate every feature available in the Batch service, nearly every Batch scenario includes portions of this workflow.



Step 1. Create **containers** in Azure Blob Storage.

Step 2. Upload task application files and input files to containers.

Step 3. Create a Batch **pool**.

3a. The pool **StartTask** downloads the task binary files (TaskApplication) to nodes as they join the pool.

Step 4. Create a Batch **job**.

Step 5. Add **tasks** to the job.

5a. The tasks are scheduled to execute on nodes.

5b. Each task downloads its input data from Azure Storage, then begins execution.

Step 6. Monitor tasks.

6a. As tasks are completed, they upload their output data to Azure Storage.

Step 7. Download task output from Storage.

As mentioned, not every Batch solution performs these exact steps, and may include many more, but the *DotNetTutorial* sample application demonstrates common processes found in a Batch solution.

Build the *DotNetTutorial* sample project

Before you can successfully run the sample, you must specify both Batch and Storage account credentials in the *DotNetTutorial* project's `Program.cs` file. If you have not done so already, open the solution in Visual Studio by double-clicking the `DotNetTutorial1.sln` solution file. Or open it from within Visual Studio by using the **File > Open > Project/Solution** menu.

Open `Program.cs` within the *DotNetTutorial* project. Then add your credentials as specified near the top of the file:

```
// Update the Batch and Storage account credential strings below with the values
// unique to your accounts. These are used when constructing connection strings
// for the Batch and Storage client objects.

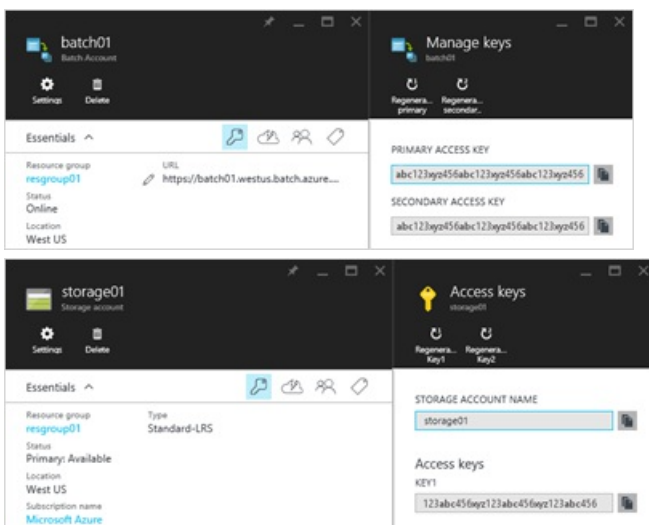
// Batch account credentials
private const string BatchAccountName = "";
private const string BatchAccountKey = "";
private const string BatchAccountUrl = "";

// Storage account credentials
private const string StorageAccountName = "";
private const string StorageAccountKey = "";
```

IMPORTANT

As mentioned above, you must currently specify the credentials for a **General purpose** storage account in Azure Storage. Your Batch applications use blob storage within the **General purpose** storage account. Do not specify the credentials for a Storage account that was created by selecting the *Blob storage* account type.

You can find your Batch and Storage account credentials within the account blade of each service in the [Azure portal](#):



Now that you've updated the project with your credentials, right-click the solution in Solution Explorer and click **Build Solution**. Confirm the restoration of any NuGet packages, if you're prompted.

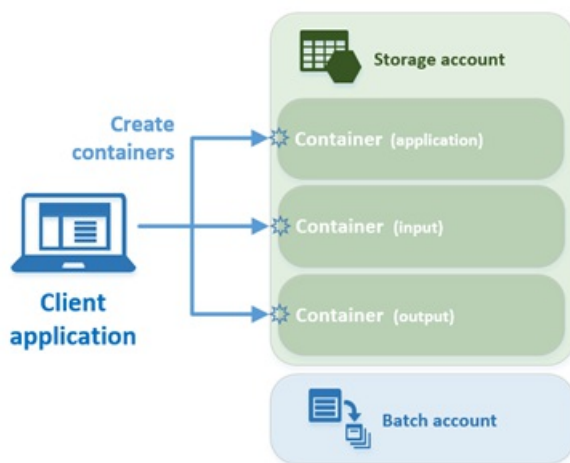
TIP

If the NuGet packages are not automatically restored, or if you see errors about a failure to restore the packages, ensure that you have the [NuGet Package Manager](#) installed. Then enable the download of missing packages. See [Enabling Package Restore During Build](#) to enable package download.

In the following sections, we break down the sample application into the steps that it performs to process a workload in the Batch service, and discuss those steps in detail. We encourage you to refer to the open solution in Visual Studio while you work your way through the rest of this article, since not every line of code in the sample is discussed.

Navigate to the top of the `MainAsync` method in the *DotNetTutorial* project's `Program.cs` file to start with Step 1. Each step below then roughly follows the progression of method calls in `MainAsync`.

Step 1: Create Storage containers



Batch includes built-in support for interacting with Azure Storage. Containers in your Storage account will provide the files needed by the tasks that run in your Batch account. The containers also provide a place to store the output data that the tasks produce. The first thing the *DotNetTutorial* client application does is create three containers in [Azure Blob Storage](#):

- **application:** This container will store the application run by the tasks, as well as any of its dependencies, such as DLLs.
- **input:** Tasks will download the data files to process from the *input* container.
- **output:** When tasks complete input file processing, they will upload the results to the *output* container.

In order to interact with a Storage account and create containers, we use the [Azure Storage Client Library for .NET](#). We create a reference to the account with `CloudStorageAccount`, and from that create a `CloudBlobClient`:

```
// Construct the Storage account connection string
string storageConnectionString = String.Format(
    "DefaultEndpointsProtocol=https;AccountName={0};AccountKey={1}",
    StorageAccountName,
    StorageAccountKey);

// Retrieve the storage account
CloudStorageAccount storageAccount =
    CloudStorageAccount.Parse(storageConnectionString);

// Create the blob client, for use in obtaining references to
// blob storage containers
CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();
```

We use the `blobClient` reference throughout the application and pass it as a parameter to several methods. An example of this is in the code block that immediately follows the above, where we call `CreateContainerIfNotExistAsync` to actually create the containers.

```
// Use the blob client to create the containers in Azure Storage if they don't
// yet exist
const string appContainerName = "application";
const string inputContainerName = "input";
const string outputContainerName = "output";
await CreateContainerIfNotExistAsync(blobClient, appContainerName);
await CreateContainerIfNotExistAsync(blobClient, inputContainerName);
await CreateContainerIfNotExistAsync(blobClient, outputContainerName);
```

```
private static async Task CreateContainerIfNotExistAsync(
    CloudBlobClient blobClient,
    string containerName)
{
    CloudBlobContainer container =
        blobClient.GetContainerReference(containerName);

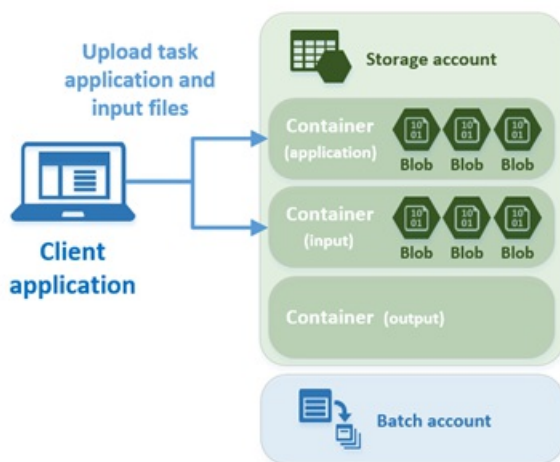
    if (await container.CreateIfNotExistsAsync())
    {
        Console.WriteLine("Container [{0}] created.", containerName);
    }
    else
    {
        Console.WriteLine("Container [{0}] exists, skipping creation.",
            containerName);
    }
}
```

Once the containers have been created, the application can now upload the files that will be used by the tasks.

TIP

[How to use Blob Storage from .NET](#) provides a good overview of working with Azure Storage containers and blobs. It should be near the top of your reading list as you start working with Batch.

Step 2: Upload task application and data files



In the file upload operation, *DotNetTutorial* first defines collections of **application** and **input** file paths as they exist on the local machine. Then it uploads these files to the containers that you created in the previous step.

```

// Paths to the executable and its dependencies that will be executed by the tasks
List<string> applicationFilePaths = new List<string>
{
    // The DotNetTutorial project includes a project reference to TaskApplication,
    // allowing us to determine the path of the task application binary dynamically
    typeof(TaskApplication.Program).Assembly.Location,
    "Microsoft.WindowsAzure.Storage.dll"
};

// The collection of data files that are to be processed by the tasks
List<string> inputFilePaths = new List<string>
{
    @"..\..\taskdata1.txt",
    @"..\..\taskdata2.txt",
    @"..\..\taskdata3.txt"
};

// Upload the application and its dependencies to Azure Storage. This is the
// application that will process the data files, and will be executed by each
// of the tasks on the compute nodes.
List<ResourceFile> applicationFiles = await UploadFilesToContainerAsync(
    blobClient,
    appContainerName,
    applicationFilePaths);

// Upload the data files. This is the data that will be processed by each of
// the tasks that are executed on the compute nodes within the pool.
List<ResourceFile> inputFiles = await UploadFilesToContainerAsync(
    blobClient,
    inputContainerName,
    inputFilePaths);

```

There are two methods in `Program.cs` that are involved in the upload process:

- `UploadFilesToContainerAsync`: This method returns a collection of [ResourceFile](#) objects (discussed below) and internally calls `UploadFileToContainerAsync` to upload each file that is passed in the *filePath*s parameter.
- `UploadFileToContainerAsync`: This is the method that actually performs the file upload and creates the [ResourceFile](#) objects. After uploading the file, it obtains a shared access signature (SAS) for the file and returns a [ResourceFile](#) object that represents it. Shared access signatures are also discussed below.

```

private static async Task<ResourceFile> UploadFileToContainerAsync(
    CloudBlobClient blobClient,
    string containerName,
    string filePath)
{
    Console.WriteLine(
        "Uploading file {0} to container [{1}]...", filePath, containerName);

    string blobName = Path.GetFileName(filePath);

    CloudBlobContainer container = blobClient.GetContainerReference(containerName);
    CloudBlockBlob blobData = container.GetBlockBlobReference(blobName);
    await blobData.UploadFromFileAsync(filePath);

    // Set the expiry time and permissions for the blob shared access signature.
    // In this case, no start time is specified, so the shared access signature
    // becomes valid immediately
    SharedAccessBlobPolicy sasConstraints = new SharedAccessBlobPolicy
    {
        SharedAccessExpiryTime = DateTime.UtcNow.AddHours(2),
        Permissions = SharedAccessBlobPermissions.Read
    };

    // Construct the SAS URL for blob
    string sasBlobToken = blobData.GetSharedAccessSignature(sasConstraints);
    string blobSasUri = String.Format("{0}{1}", blobData.Uri, sasBlobToken);

    return new ResourceFile(blobSasUri, blobName);
}

```

ResourceFiles

A [ResourceFile](#) provides tasks in Batch with the URL to a file in Azure Storage that is downloaded to a compute node before that task is run. The [ResourceFile.BlobSource](#) property specifies the full URL of the file as it exists in Azure Storage. The URL may also include a shared access signature (SAS) that provides secure access to the file. Most tasks types within Batch .NET include a *ResourceFiles* property, including:

- [CloudTask](#)
- [StartTask](#)
- [JobPreparationTask](#)
- [JobReleaseTask](#)

The DotNetTutorial sample application does not use the JobPreparationTask or JobReleaseTask task types, but you can read more about them in [Run job preparation and completion tasks on Azure Batch compute nodes](#).

Shared access signature (SAS)

Shared access signatures are strings which—when included as part of a URL—provide secure access to containers and blobs in Azure Storage. The DotNetTutorial application uses both blob and container shared access signature URLs, and demonstrates how to obtain these shared access signature strings from the Storage service.

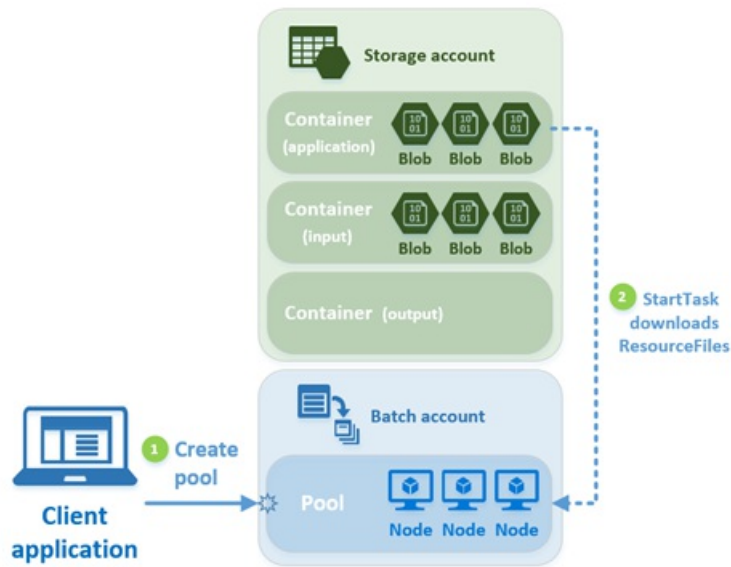
- **Blob shared access signatures:** The pool's StartTask in DotNetTutorial uses blob shared access signatures when it downloads the application binaries and input data files from Storage (see Step #3 below). The `UploadFileToContainerAsync` method in DotNetTutorial's `Program.cs` contains the code that obtains each blob's shared access signature. It does so by calling [CloudBlob.GetSharedAccessSignature](#).
- **Container shared access signatures:** As each task finishes its work on the compute node, it uploads its output file to the *output* container in Azure Storage. To do so, TaskApplication uses a container shared access signature that provides write access to the container as part of the path when it uploads the file. Obtaining the container shared access signature is done in a similar fashion as when obtaining the blob shared access

signature. In *DotNetTutorial*, you will find that the `GetContainerSasUrl` helper method calls [CloudBlobContainer.GetSharedAccessSignature](#) to do so. You'll read more about how *TaskApplication* uses the container shared access signature in "Step 6: Monitor Tasks."

TIP

Check out the two-part series on shared access signatures, [Part 1: Understanding the shared access signature \(SAS\) model](#) and [Part 2: Create and use a shared access signature \(SAS\) with Blob storage](#), to learn more about providing secure access to data in your Storage account.

Step 3: Create Batch pool



A Batch **pool** is a collection of compute nodes (virtual machines) on which Batch executes a job's tasks.

After it uploads the application and data files to the Storage account, *DotNetTutorial* starts its interaction with the Batch service by using the Batch .NET library. To do so, a [BatchClient](#) is first created:

```
BatchSharedKeyCredentials cred = new BatchSharedKeyCredentials(  
    BatchAccountUrl,  
    BatchAccountName,  
    BatchAccountKey);  
  
using (BatchClient batchClient = BatchClient.Open(cred))  
{  
    ...  
}
```

Next, a pool of compute nodes is created in the Batch account with a call to `CreatePoolIfNotExistsAsync`. `CreatePoolIfNotExistsAsync` uses the [BatchClient.PoolOperations.CreatePool](#) method to create a pool in the Batch service.

```

private static async Task CreatePoolIfNotExistAsync(BatchClient batchClient, string poolId,
IList<ResourceFile> resourceFiles)
{
    CloudPool pool = null;
    try
    {
        Console.WriteLine("Creating pool [{0}]...", poolId);

        // Create the unbound pool. Until we call CloudPool.Commit() or CommitAsync(), no pool is actually
        // created in the
        // Batch service. This CloudPool instance is therefore considered "unbound," and we can modify its
        // properties.
        pool = batchClient.PoolOperations.CreatePool(
            poolId: poolId,
            targetDedicated: 3, // 3 compute nodes
            virtualMachineSize: "small", // single-core, 1.75
            GB memory, 225 GB disk
            cloudServiceConfiguration: new CloudServiceConfiguration(osFamily: "4")); // Windows Server
            2012 R2

        // Create and assign the StartTask that will be executed when compute nodes join the pool.
        // In this case, we copy the StartTask's resource files (that will be automatically downloaded
        // to the node by the StartTask) into the shared directory that all tasks will have access to.
        pool.StartTask = new StartTask
        {
            // Specify a command line for the StartTask that copies the task application files to the
            // node's shared directory. Every compute node in a Batch pool is configured with a number
            // of pre-defined environment variables that can be referenced by commands or applications
            // run by tasks.

            // Since a successful execution of robocopy can return a non-zero exit code (e.g. 1 when one or
            // more files were successfully copied) we need to manually exit with a 0 for Batch to recognize
            // StartTask execution success.
            CommandLine = "cmd /c (robocopy %AZ_BATCH_TASK_WORKING_DIR% %AZ_BATCH_NODE_SHARED_DIR%) ^& IF
%ERRORLEVEL% LEQ 1 exit 0",
            ResourceFiles = resourceFiles,
            WaitForSuccess = true
        };

        await pool.CommitAsync();
    }
    catch (BatchException be)
    {
        // Swallow the specific error code PoolExists since that is expected if the pool already exists
        if (be.RequestInformation?.BatchError != null && be.RequestInformation.BatchError.Code ==
BatchErrorCodeStrings.PoolExists)
        {
            Console.WriteLine("The pool {0} already existed when we tried to create it", poolId);
        }
        else
        {
            throw; // Any other exception is unexpected
        }
    }
}

```

When you create a pool with [CreatePool](#), you specify several parameters such as the number of compute nodes, the [size of the nodes](#), and the nodes' operating system. In *DotNetTutorial*, we use [CloudServiceConfiguration](#) to specify Windows Server 2012 R2 from [Cloud Services](#). However, by specifying a [VirtualMachineConfiguration](#) instead, you can create pools of nodes created from Marketplace images, which includes both Windows and Linux images—see [Provision Linux compute nodes in Azure Batch pools](#) for more information.

IMPORTANT

You are charged for compute resources in Batch. To minimize costs, you can lower `targetDedicated` to 1 before you run the sample.

Along with these physical node properties, you may also specify a [StartTask](#) for the pool. The StartTask executes on each node as that node joins the pool, and each time a node is restarted. The StartTask is especially useful for installing applications on compute nodes prior to the execution of tasks. For example, if your tasks process data by using Python scripts, you could use a StartTask to install Python on the compute nodes.

In this sample application, the StartTask copies the files that it downloads from Storage (which are specified by using the [StartTask.ResourceFiles](#) property) from the StartTask working directory to the shared directory that *all* tasks running on the node can access. Essentially, this copies `TaskApplication.exe` and its dependencies to the shared directory on each node as the node joins the pool, so that any tasks that run on the node can access it.

TIP

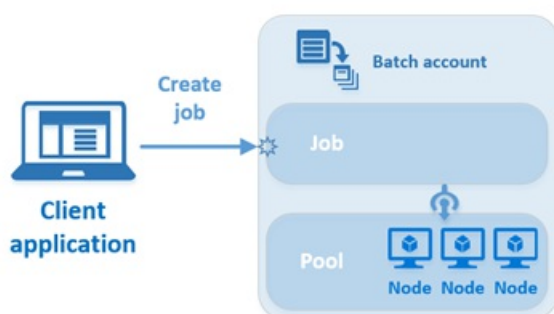
The **application packages** feature of Azure Batch provides another way to get your application onto the compute nodes in a pool. See [Application deployment with Azure Batch application packages](#) for details.

Also notable in the code snippet above is the use of two environment variables in the *CommandLine* property of the StartTask: `%AZ_BATCH_TASK_WORKING_DIR%` and `%AZ_BATCH_NODE_SHARED_DIR%`. Each compute node within a Batch pool is automatically configured with several environment variables that are specific to Batch. Any process that is executed by a task has access to these environment variables.

TIP

To find out more about the environment variables that are available on compute nodes in a Batch pool, and information on task working directories, see the [Environment settings for tasks](#) and [Files and directories](#) sections in the [Batch feature overview for developers](#).

Step 4: Create Batch job



A Batch **job** is a collection of tasks, and is associated with a pool of compute nodes. The tasks in a job execute on the associated pool's compute nodes.

You can use a job not only for organizing and tracking tasks in related workloads, but also for imposing certain constraints--such as the maximum runtime for the job (and by extension, its tasks) as well as job priority in relation to other jobs in the Batch account. In this example, however, the job is associated only with the pool that was created in step #3. No additional properties are configured.

All Batch jobs are associated with a specific pool. This association indicates which nodes the job's tasks will execute on. You specify this by using the [CloudJob.PoolInformation](#) property, as shown in the code snippet

below.

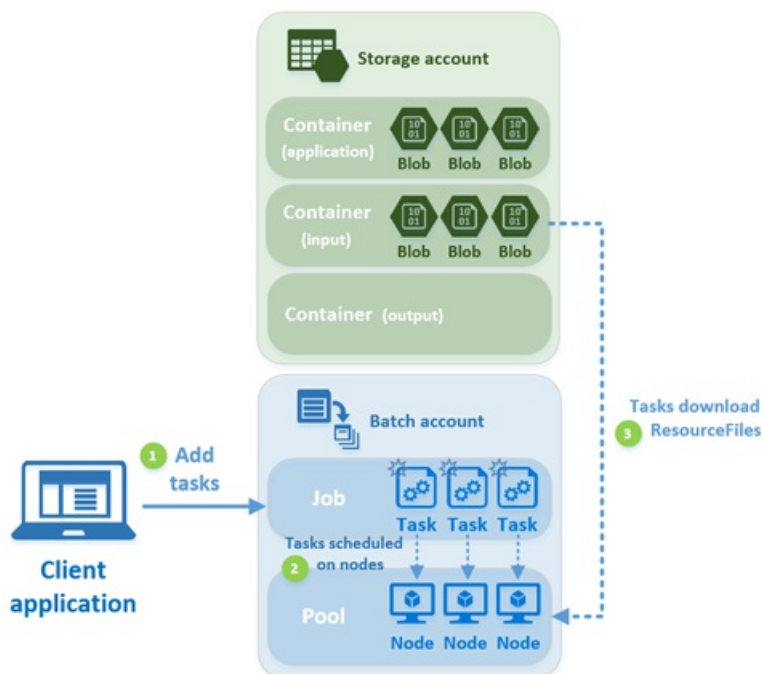
```
private static async Task CreateJobAsync(
    BatchClient batchClient,
    string jobId,
    string poolId)
{
    Console.WriteLine("Creating job [{0}]...", jobId);

    CloudJob job = batchClient.JobOperations.CreateJob();
    job.Id = jobId;
    job.PoolInformation = new PoolInformation { PoolId = poolId };

    await job.CommitAsync();
}
```

Now that a job has been created, tasks are added to perform the work.

Step 5: Add tasks to job



(1) Tasks are added to the job, (2) the tasks are scheduled to run on nodes, and (3) the tasks download the data files to process

Batch **tasks** are the individual units of work that execute on the compute nodes. A task has a command line and runs the scripts or executables that you specify in that command line.

To actually perform work, tasks must be added to a job. Each **CloudTask** is configured by using a command-line property and **ResourceFiles** (as with the pool's **StartTask**) that the task downloads to the node before its command line is automatically executed. In the *DotNetTutorial* sample project, each task processes only one file. Thus, its **ResourceFiles** collection contains a single element.

```

private static async Task<List<CloudTask>> AddTasksAsync(
    BatchClient batchClient,
    string jobId,
    List<ResourceFile> inputFiles,
    string outputContainerSasUrl)
{
    Console.WriteLine("Adding {0} tasks to job [{1}]...", inputFiles.Count, jobId);

    // Create a collection to hold the tasks that we'll be adding to the job
    List<CloudTask> tasks = new List<CloudTask>();

    // Create each of the tasks. Because we copied the task application to the
    // node's shared directory with the pool's StartTask, we can access it via
    // the shared directory on the node that the task runs on.
    foreach (ResourceFile inputFile in inputFiles)
    {
        string taskId = "topNtask" + inputFiles.IndexOf(inputFile);
        string taskCommandLine = String.Format(
            "cmd /c %AZ_BATCH_NODE_SHARED_DIR%\\TaskApplication.exe {0} 3 \"{1}\"",
            inputFile.FilePath,
            outputContainerSasUrl);

        CloudTask task = new CloudTask(taskId, taskCommandLine);
        task.ResourceFiles = new List<ResourceFile> { inputFile };
        tasks.Add(task);
    }

    // Add the tasks as a collection, as opposed to issuing a separate AddTask call
    // for each. Bulk task submission helps to ensure efficient underlying API calls
    // to the Batch service.
    await batchClient.JobOperations.AddTaskAsync(jobId, tasks);

    return tasks;
}

```

IMPORTANT

When they access environment variables such as `%AZ_BATCH_NODE_SHARED_DIR%` or execute an application not found in the node's `PATH`, task command lines must be prefixed with `cmd /c`. This will explicitly execute the command interpreter and instruct it to terminate after carrying out your command. This requirement is unnecessary if your tasks execute an application in the node's `PATH` (such as *robocopy.exe* or *powershell.exe*) and no environment variables are used.

Within the `foreach` loop in the code snippet above, you can see that the command line for the task is constructed such that three command-line arguments are passed to *TaskApplication.exe*:

1. The **first argument** is the path of the file to process. This is the local path to the file as it exists on the node. When the `ResourceFile` object in `UploadFileToContainerAsync` was first created above, the file name was used for this property (as a parameter to the `ResourceFile` constructor). This indicates that the file can be found in the same directory as *TaskApplication.exe*.
2. The **second argument** specifies that the top *N* words should be written to the output file. In the sample, this is hard-coded so that the top three words are written to the output file.
3. The **third argument** is the shared access signature (SAS) that provides write access to the **output** container in Azure Storage. *TaskApplication.exe* uses this shared access signature URL when it uploads the output file to Azure Storage. You can find the code for this in the `UploadFileToContainer` method in the `TaskApplication` project's `Program.cs` file:

```
// NOTE: From project TaskApplication Program.cs

private static void UploadFileToContainer(string filePath, string containerSas)
{
    string blobName = Path.GetFileName(filePath);

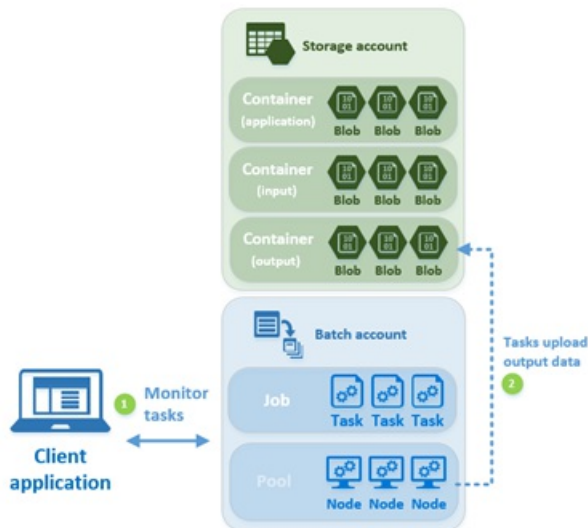
    // Obtain a reference to the container using the SAS URI.
    CloudBlobContainer container = new CloudBlobContainer(new Uri(containerSas));

    // Upload the file (as a new blob) to the container
    try
    {
        CloudBlockBlob blob = container.GetBlockBlobReference(blobName);
        blob.UploadFromFile(filePath);

        Console.WriteLine("Write operation succeeded for SAS URL " + containerSas);
        Console.WriteLine();
    }
    catch (StorageException e)
    {
        Console.WriteLine("Write operation failed for SAS URL " + containerSas);
        Console.WriteLine("Additional error information: " + e.Message);
        Console.WriteLine();

        // Indicate that a failure has occurred so that when the Batch service
        // sets the CloudTask.ExecutionInformation.ExitCode for the task that
        // executed this application, it properly indicates that there was a
        // problem with the task.
        Environment.ExitCode = -1;
    }
}
}
```

Step 6: Monitor tasks



The client application (1) monitors the tasks for completion and success status, and (2) the tasks upload result data to Azure Storage

When tasks are added to a job, they are automatically queued and scheduled for execution on compute nodes within the pool associated with the job. Based on the settings you specify, Batch handles all task queuing, scheduling, retrying, and other task administration duties for you.

There are many approaches to monitoring task execution. DotNetTutorial shows a simple example that reports only on completion and task failure or success states. Within the `MonitorTasks` method in DotNetTutorial's `Program.cs`, there are three Batch .NET concepts that warrant discussion. They are listed below in their order of

appearance:

1. **ODATADetailLevel**: Specifying [ODATADetailLevel](#) in list operations (such as obtaining a list of a job's tasks) is essential in ensuring Batch application performance. Add [Query the Azure Batch service efficiently](#) to your reading list if you plan on doing any sort of status monitoring within your Batch applications.
2. **TaskStateMonitor**: [TaskStateMonitor](#) provides Batch .NET applications with helper utilities for monitoring task states. In `MonitorTasks`, *DotNetTutorial* waits for all tasks to reach [TaskState.Completed](#) within a time limit. Then it terminates the job.
3. **TerminateJobAsync**: Terminating a job with [JobOperations.TerminateJobAsync](#) (or the blocking `JobOperations.TerminateJob`) marks that job as completed. It is essential to do so if your Batch solution uses a [JobReleaseTask](#). This is a special type of task, which is described in [Job preparation and completion tasks](#).

The `MonitorTasks` method from *DotNetTutorial*'s `Program.cs` appears below:

```
private static async Task<bool> MonitorTasks(
    BatchClient batchClient,
    string jobId,
    TimeSpan timeout)
{
    bool allTasksSuccessful = true;
    const string successMessage = "All tasks reached state Completed.";
    const string failureMessage = "One or more tasks failed to reach the Completed state within the timeout period.";

    // Obtain the collection of tasks currently managed by the job. Note that we use
    // a detail level to specify that only the "id" property of each task should be
    // populated. Using a detail level for all list operations helps to lower
    // response time from the Batch service.
    ODATADetailLevel detail = new ODATADetailLevel(selectClause: "id");
    List<CloudTask> tasks =
        await batchClient.JobOperations.ListTasks(jobId, detail).ToListAsync();

    Console.WriteLine("Awaiting task completion, timeout in {0}...",
        timeout.ToString());

    // We use a TaskStateMonitor to monitor the state of our tasks. In this case, we
    // will wait for all tasks to reach the Completed state.
    TaskStateMonitor taskStateMonitor
        = batchClient.Utilities.CreateTaskStateMonitor();

    try
    {
        await taskStateMonitor.WhenAll(tasks, TaskState.Completed, timeout);
    }
    catch (TimeoutException)
    {
        await batchClient.JobOperations.TerminateJobAsync(jobId, failureMessage);
        Console.WriteLine(failureMessage);
        return false;
    }

    await batchClient.JobOperations.TerminateJobAsync(jobId, successMessage);

    // All tasks have reached the "Completed" state, however, this does not
    // guarantee all tasks completed successfully. Here we further check each task's
    // ExecutionInfo property to ensure that it did not encounter a scheduling error
    // or return a non-zero exit code.

    // Update the detail level to populate only the task id and executionInfo
    // properties. We refresh the tasks below, and need only this information for
    // each task.
    detail.SelectClause = "id, executionInfo";

    foreach (CloudTask task in tasks)
    {
```

```

// Populate the task's properties with the latest info from the
// Batch service
await task.RefreshAsync(detail);

if (task.ExecutionInformation.SchedulingError != null)
{
    // A scheduling error indicates a problem starting the task on the node.
    // It is important to note that the task's state can be "Completed," yet
    // still have encountered a scheduling error.

    allTasksSuccessful = false;

    Console.WriteLine("WARNING: Task [{0}] encountered a scheduling error: {1}",
        task.Id,
        task.ExecutionInformation.SchedulingError.Message);
}
else if (task.ExecutionInformation.ExitCode != 0)
{
    // A non-zero exit code may indicate that the application executed by
    // the task encountered an error during execution. As not every
    // application returns non-zero on failure by default (e.g. robocopy),
    // your implementation of error checking may differ from this example.

    allTasksSuccessful = false;

    Console.WriteLine("WARNING: Task [{0}] returned a non-zero exit code - this may indicate task
    execution or completion failure.", task.Id);
}

if (allTasksSuccessful)
{
    Console.WriteLine("Success! All tasks completed successfully within the specified timeout period.");
}

return allTasksSuccessful;
}

```

Step 7: Download task output



Now that the job is completed, the output from the tasks can be downloaded from Azure Storage. This is done with a call to `DownloadBlobsFromContainerAsync` in *DotNetTutorial's* `Program.cs`:

```
private static async Task DownloadBlobsFromContainerAsync(
    CloudBlobClient blobClient,
    string containerName,
    string directoryPath)
{
    Console.WriteLine("Downloading all files from container [{0}]...", containerName);

    // Retrieve a reference to a previously created container
    CloudBlobContainer container = blobClient.GetContainerReference(containerName);

    // Get a flat listing of all the block blobs in the specified container
    foreach (IListBlobItem item in container.ListBlobs(
        prefix: null,
        useFlatBlobListing: true))
    {
        // Retrieve reference to the current blob
        CloudBlob blob = (CloudBlob)item;

        // Save blob contents to a file in the specified folder
        string localOutputFile = Path.Combine(directoryPath, blob.Name);
        await blob.DownloadToFileAsync(localOutputFile, FileMode.Create);
    }

    Console.WriteLine("All files downloaded to {0}", directoryPath);
}
```

NOTE

The call to `DownloadBlobsFromContainerAsync` in the *DotNetTutorial* application specifies that the files should be downloaded to your `%TEMP%` folder. Feel free to modify this output location.

Step 8: Delete containers

Because you are charged for data that resides in Azure Storage, it's always a good idea to remove blobs that are no longer needed for your Batch jobs. In *DotNetTutorial*'s `Program.cs`, this is done with three calls to the helper method `DeleteContainerAsync`:

```
// Clean up Storage resources
await DeleteContainerAsync(blobClient, appContainerName);
await DeleteContainerAsync(blobClient, inputContainerName);
await DeleteContainerAsync(blobClient, outputContainerName);
```

The method itself merely obtains a reference to the container, and then calls [CloudBlobContainer.DeleteIfExistsAsync](#):

```
private static async Task DeleteContainerAsync(
    CloudBlobClient blobClient,
    string containerName)
{
    CloudBlobContainer container = blobClient.GetContainerReference(containerName);

    if (await container.DeleteIfExistsAsync())
    {
        Console.WriteLine("Container [{0}] deleted.", containerName);
    }
    else
    {
        Console.WriteLine("Container [{0}] does not exist, skipping deletion.",
            containerName);
    }
}
```

Step 9: Delete the job and the pool

In the final step, you're prompted to delete the job and the pool that were created by the DotNetTutorial application. Although you're not charged for jobs and tasks themselves, you *are* charged for compute nodes. Thus, we recommend that you allocate nodes only as needed. Deleting unused pools can be part of your maintenance process.

The BatchClient's [JobOperations](#) and [PoolOperations](#) both have corresponding deletion methods, which are called if the user confirms deletion:

```
// Clean up the resources we've created in the Batch account if the user so chooses
Console.WriteLine();
Console.WriteLine("Delete job? [yes] no");
string response = Console.ReadLine().ToLower();
if (response != "n" && response != "no")
{
    await batchClient.JobOperations.DeleteJobAsync(JobId);
}

Console.WriteLine("Delete pool? [yes] no");
response = Console.ReadLine();
if (response != "n" && response != "no")
{
    await batchClient.PoolOperations.DeletePoolAsync(PoolId);
}
```

IMPORTANT

Keep in mind that you are charged for compute resources—deleting unused pools will minimize cost. Also, be aware that deleting a pool deletes all compute nodes within that pool, and that any data on the nodes will be unrecoverable after the pool is deleted.

Run the *DotNetTutorial* sample

When you run the sample application, the console output will be similar to the following. During execution, you will experience a pause at `Awaiting task completion, timeout in 00:30:00...` while the pool's compute nodes are started. Use the [Azure portal](#) to monitor your pool, compute nodes, job, and tasks during and after execution. Use the [Azure portal](#) or the [Azure Storage Explorer](#) to view the Storage resources (containers and blobs) that are created by the application.

Typical execution time is **approximately 5 minutes** when you run the application in its default configuration.


```
Sample start: 1/8/2016 09:42:58 AM

Container [application] created.
Container [input] created.
Container [output] created.
Uploading file C:\repos\azure-batch-samples\CSharp\ArticleProjects\DotNetTutorial\bin\Debug\TaskApplication.exe to container [application]...
Uploading file Microsoft.WindowsAzure.Storage.dll to container [application]...
Uploading file ..\..\taskdata1.txt to container [input]...
Uploading file ..\..\taskdata2.txt to container [input]...
Uploading file ..\..\taskdata3.txt to container [input]...
Creating pool [DotNetTutorialPool]...
Creating job [DotNetTutorialJob]...
Adding 3 tasks to job [DotNetTutorialJob]...
Awaiting task completion, timeout in 00:30:00...
Success! All tasks completed successfully within the specified timeout period.
Downloading all files from container [output]...
All files downloaded to C:\Users\USERNAME\AppData\Local\Temp
Container [application] deleted.
Container [input] deleted.
Container [output] deleted.

Sample end: 1/8/2016 09:47:47 AM
Elapsed time: 00:04:48.5358142

Delete job? [yes] no: yes
Delete pool? [yes] no: yes

Sample complete, hit ENTER to exit...
```

Next steps

Feel free to make changes to *DotNetTutorial* and *TaskApplication* to experiment with different compute scenarios. For example, try adding an execution delay to *TaskApplication*, such as with [Thread.Sleep](#), to simulate long-running tasks and monitor them in the portal. Try adding more tasks or adjusting the number of compute nodes. Add logic to check for and allow the use of an existing pool to speed execution time (*hint*: check out `ArticleHelpers.cs` in the [Microsoft.Azure.Batch.Samples.Common](#) project in [azure-batch-samples](#)).

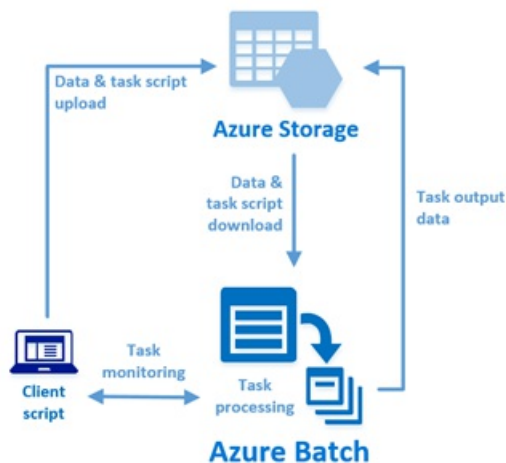
Now that you're familiar with the basic workflow of a Batch solution, it's time to dig in to the additional features of the Batch service.

- Review the [Overview of Azure Batch features](#) article, which we recommend if you're new to the service.
- Start on the other Batch development articles under **Development in-depth** in the [Batch learning path](#).
- Check out a different implementation of processing the "top N words" workload by using Batch in the [TopNWords](#) sample.

Get started with the Batch SDK for Python

3/13/2017 • 22 min to read • [Edit Online](#)

Learn the basics of [Azure Batch](#) and the [Batch Python](#) client as we discuss a small Batch application written in Python. We look at how two sample scripts use the Batch service to process a parallel workload on Linux virtual machines in the cloud, and how they interact with [Azure Storage](#) for file staging and retrieval. You'll learn a common Batch application workflow and gain a base understanding of the major components of Batch such as jobs, tasks, pools, and compute nodes.



Prerequisites

This article assumes that you have a working knowledge of Python and familiarity with Linux. It also assumes that you're able to satisfy the account creation requirements that are specified below for Azure and the Batch and Storage services.

Accounts

- **Azure account:** If you don't already have an Azure subscription, [create a free Azure account](#).
- **Batch account:** Once you have an Azure subscription, [create an Azure Batch account](#).
- **Storage account:** See [Create a storage account](#) in [About Azure storage accounts](#).

Code sample

The Python tutorial [code sample](#) is one of the many Batch code samples found in the [azure-batch-samples](#) repository on GitHub. You can download all the samples by clicking **Clone or download** > **Download ZIP** on the repository home page, or by clicking the [azure-batch-samples-master.zip](#) direct download link. Once you've extracted the contents of the ZIP file, the two scripts for this tutorial are found in the `article_samples` directory:

```
/azure-batch-samples/Python/Batch/article_samples/python_tutorial_client.py
```

```
/azure-batch-samples/Python/Batch/article_samples/python_tutorial_task.py
```

Python environment

To run the `python_tutorial_client.py` sample script on your local workstation, you need a **Python interpreter** compatible with version **2.7** or **3.3+**. The script has been tested on both Linux and Windows.

cryptography dependencies

You must install the dependencies for the [cryptography](#) library, required by the `azure-batch` and `azure-storage` Python packages. Perform one of the following operations appropriate for your platform, or refer to the [cryptography installation](#) details for more information:

- Ubuntu

```
apt-get update && apt-get install -y build-essential libssl-dev libffi-dev libpython-dev python-dev
```

- CentOS

```
yum update && yum install -y gcc openssl-devel libffi-devel python-devel
```

- SLES/OpenSUSE

```
zypper ref && zypper -n in libopenssl-devel libffi48-devel python-devel
```

- Windows

```
pip install cryptography
```

NOTE

If installing for Python 3.3+ on Linux, use the python3 equivalents for the Python dependencies. For example, on Ubuntu:

```
apt-get update && apt-get install -y build-essential libssl-dev libffi-dev libpython3-dev python3-dev
```

Azure packages

Next, install the **Azure Batch** and **Azure Storage** Python packages. You can install both packages by using **pip** and the *requirements.txt* found here:

```
/azure-batch-samples/Python/Batch/requirements.txt
```

Issue following **pip** command to install the Batch and Storage packages:

```
pip install -r requirements.txt
```

Or, you can install the [azure-batch](#) and [azure-storage](#) Python packages manually:

```
pip install azure-batch
```

```
pip install azure-storage
```

TIP

If you are using an unprivileged account, you may need to prefix your commands with `sudo`. For example,

```
sudo pip install -r requirements.txt
```

For more information on installing Python packages, see [Installing Packages on python.org](#).

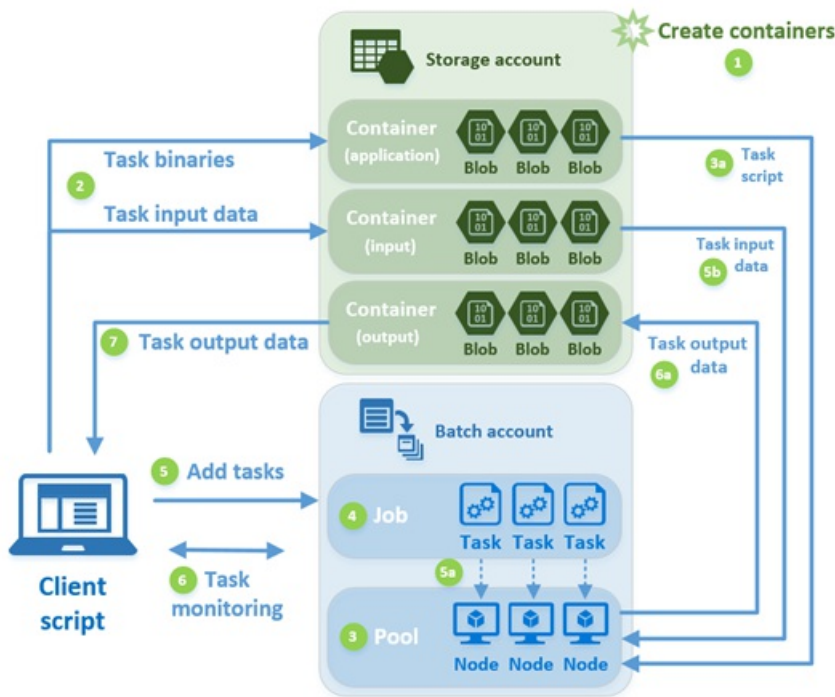
Batch Python tutorial code sample

The Batch Python tutorial code sample consists of two Python scripts and a few data files.

- **python_tutorial_client.py**: Interacts with the Batch and Storage services to execute a parallel workload on compute nodes (virtual machines). The *python_tutorial_client.py* script runs on your local workstation.
- **python_tutorial_task.py**: The script that runs on compute nodes in Azure to perform the actual work. In the sample, *python_tutorial_task.py* parses the text in a file downloaded from Azure Storage (the input file). Then it produces a text file (the output file) that contains a list of the top three words that appear in the input file. After it creates the output file, *python_tutorial_task.py* uploads the file to Azure Storage. This makes it available for download to the client script running on your workstation. The *python_tutorial_task.py* script runs in parallel on multiple compute nodes in the Batch service.
- **./data/taskdata*.txt**: These three text files provide the input for the tasks that run on the compute nodes.

The following diagram illustrates the primary operations that are performed by the client and task scripts. This

basic workflow is typical of many compute solutions that are created with Batch. While it does not demonstrate every feature available in the Batch service, nearly every Batch scenario includes portions of this workflow.



Step 1. Create **containers** in Azure Blob Storage.

Step 2. Upload task script and input files to containers.

Step 3. Create a Batch **pool**.

3a. The pool **StartTask** downloads the task script (python_tutorial_task.py) to nodes as they join the pool.

Step 4. Create a Batch **job**.

Step 5. Add **tasks** to the job.

5a. The tasks are scheduled to execute on nodes.

5b. Each task downloads its input data from Azure Storage, then begins execution.

Step 6. Monitor tasks.

6a. As tasks are completed, they upload their output data to Azure Storage.

Step 7. Download task output from Storage.

As mentioned, not every Batch solution performs these exact steps, and may include many more, but this sample demonstrates common processes found in a Batch solution.

Prepare client script

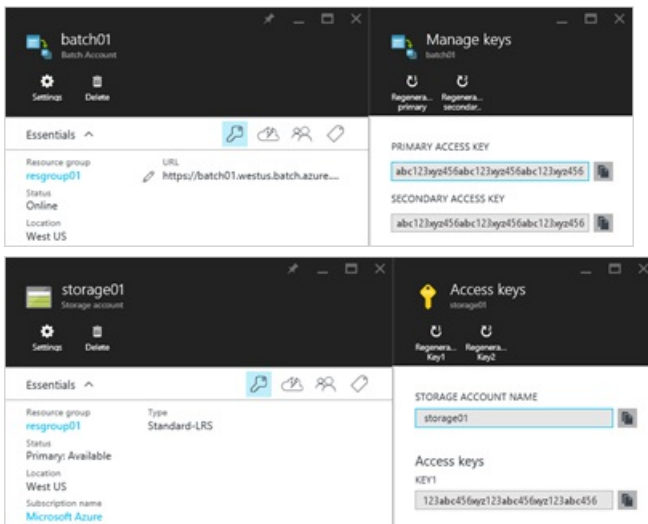
Before you run the sample, add your Batch and Storage account credentials to *python_tutorial_client.py*. If you have not done so already, open the file in your favorite editor and update the following lines with your credentials.

```
# Update the Batch and Storage account credential strings below with the values
# unique to your accounts. These are used when constructing connection strings
# for the Batch and Storage client objects.

# Batch account credentials
batch_account_name = "";
batch_account_key = "";
batch_account_url = "";

# Storage account credentials
storage_account_name = "";
storage_account_key = "";
```

You can find your Batch and Storage account credentials within the account blade of each service in the [Azure portal](#):

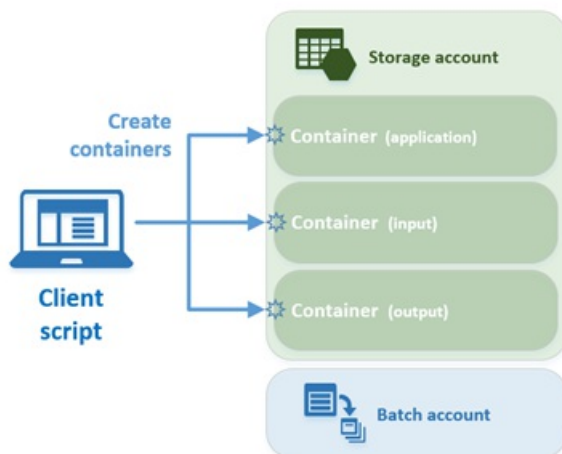


In the following sections, we analyze the steps used by the scripts to process a workload in the Batch service. We encourage you to refer regularly to the scripts in your editor while you work your way through the rest of the article.

Navigate to the following line in **python_tutorial_client.py** to start with Step 1:

```
if __name__ == '__main__':
```

Step 1: Create Storage containers



Batch includes built-in support for interacting with Azure Storage. Containers in your Storage account will provide the files needed by the tasks that run in your Batch account. The containers also provide a place to store the output data that the tasks produce. The first thing the *python_tutorial_client.py* script does is create three containers in [Azure Blob Storage](#):

- **application:** This container will store the Python script run by the tasks, *python_tutorial_task.py*.
- **input:** Tasks will download the data files to process from the *input* container.
- **output:** When tasks complete input file processing, they will upload the results to the *output* container.

In order to interact with a Storage account and create containers, we use the [azure-storage](#) package to create a [BlockBlobService](#) object--the "blob client." We then create three containers in the Storage account using the blob client.

```
# Create the blob client, for use in obtaining references to
# blob storage containers and uploading files to containers.
blob_client = azureblob.BlockBlobService(
    account_name=_STORAGE_ACCOUNT_NAME,
    account_key=_STORAGE_ACCOUNT_KEY)

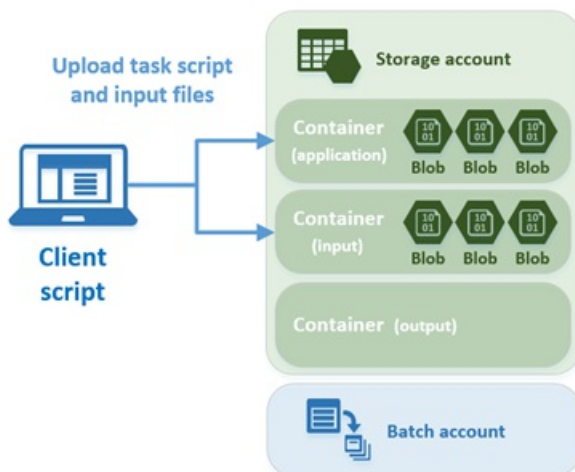
# Use the blob client to create the containers in Azure Storage if they
# don't yet exist.
app_container_name = 'application'
input_container_name = 'input'
output_container_name = 'output'
blob_client.create_container(app_container_name, fail_on_exist=False)
blob_client.create_container(input_container_name, fail_on_exist=False)
blob_client.create_container(output_container_name, fail_on_exist=False)
```

Once the containers have been created, the application can now upload the files that will be used by the tasks.

TIP

[How to use Azure Blob storage from Python](#) provides a good overview of working with Azure Storage containers and blobs. It should be near the top of your reading list as you start working with Batch.

Step 2: Upload task script and data files



In the file upload operation, *python_tutorial_client.py* first defines collections of **application** and **input** file paths as they exist on the local machine. Then it uploads these files to the containers that you created in the previous step.

```

# Paths to the task script. This script will be executed by the tasks that
# run on the compute nodes.
application_file_paths = [os.path.realpath('python_tutorial_task.py')]

# The collection of data files that are to be processed by the tasks.
input_file_paths = [os.path.realpath('./data/taskdata1.txt'),
                    os.path.realpath('./data/taskdata2.txt'),
                    os.path.realpath('./data/taskdata3.txt')]

# Upload the application script to Azure Storage. This is the script that
# will process the data files, and is executed by each of the tasks on the
# compute nodes.
application_files = [
    upload_file_to_container(blob_client, app_container_name, file_path)
    for file_path in application_file_paths]

# Upload the data files. This is the data that will be processed by each of
# the tasks executed on the compute nodes in the pool.
input_files = [
    upload_file_to_container(blob_client, input_container_name, file_path)
    for file_path in input_file_paths]

```

Using list comprehension, the `upload_file_to_container` function is called for each file in the collections, and two [ResourceFile](#) collections are populated. The `upload_file_to_container` function appears below:

```

def upload_file_to_container(block_blob_client, container_name, file_path):
    """
    Uploads a local file to an Azure Blob storage container.

    :param block_blob_client: A blob service client.
    :type block_blob_client: `azure.storage.blob.BlockBlobService`
    :param str container_name: The name of the Azure Blob storage container.
    :param str file_path: The local path to the file.
    :rtype: `azure.batch.models.ResourceFile`
    :return: A ResourceFile initialized with a SAS URL appropriate for Batch
    tasks.
    """
    blob_name = os.path.basename(file_path)

    print('Uploading file {} to container [{}]...'.format(file_path,
                                                         container_name))

    block_blob_client.create_blob_from_path(container_name,
                                             blob_name,
                                             file_path)

    sas_token = block_blob_client.generate_blob_shared_access_signature(
        container_name,
        blob_name,
        permission=azureblob.BlobPermissions.READ,
        expiry=datetime.datetime.utcnow() + datetime.timedelta(hours=2))

    sas_url = block_blob_client.make_blob_url(container_name,
                                              blob_name,
                                              sas_token=sas_token)

    return batchmodels.ResourceFile(file_path=blob_name,
                                    blob_source=sas_url)

```

ResourceFiles

A [ResourceFile](#) provides tasks in Batch with the URL to a file in Azure Storage that is downloaded to a compute node before that task is run. The [ResourceFile](#).**blob_source** property specifies the full URL of the file as it exists in Azure Storage. The URL may also include a shared access signature (SAS) that provides secure access to the file.

Most task types in Batch include a *ResourceFiles* property, including:

- [CloudTask](#)
- [StartTask](#)
- [JobPreparationTask](#)
- [JobReleaseTask](#)

This sample does not use the `JobPreparationTask` or `JobReleaseTask` task types, but you can read more about them in [Run job preparation and completion tasks on Azure Batch compute nodes](#).

Shared access signature (SAS)

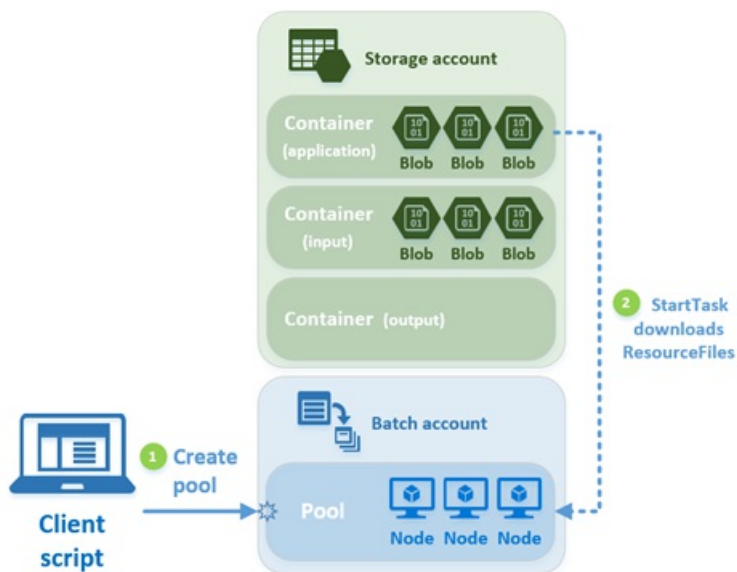
Shared access signatures are strings that provide secure access to containers and blobs in Azure Storage. The *python_tutorial_client.py* script uses both blob and container shared access signatures, and demonstrates how to obtain these shared access signature strings from the Storage service.

- **Blob shared access signatures:** The pool's `StartTask` uses blob shared access signatures when it downloads the task script and input data files from Storage (see [Step #3](#) below). The `upload_file_to_container` function in *python_tutorial_client.py* contains the code that obtains each blob's shared access signature. It does so by calling `BlockBlobService.make_blob_url` in the Storage module.
- **Container shared access signature:** As each task finishes its work on the compute node, it uploads its output file to the *output* container in Azure Storage. To do so, *python_tutorial_task.py* uses a container shared access signature that provides write access to the container. The `get_container_sas_token` function in *python_tutorial_client.py* obtains the container's shared access signature, which is then passed as a command-line argument to the tasks. Step #5, [Add tasks to a job](#), discusses the usage of the container SAS.

TIP

Check out the two-part series on shared access signatures, [Part 1: Understanding the SAS model](#) and [Part 2: Create and use a SAS with the Blob service](#), to learn more about providing secure access to data in your Storage account.

Step 3: Create Batch pool



A Batch **pool** is a collection of compute nodes (virtual machines) on which Batch executes a job's tasks.

After it uploads the task script and data files to the Storage account, *python_tutorial_client.py* starts its interaction with the Batch service by using the Batch Python module. To do so, a [BatchServiceClient](#) is created:


```
# Create a Batch service client. We'll now be interacting with the Batch
# service in addition to Storage.
credentials = batchauth.SharedKeyCredentials(_BATCH_ACCOUNT_NAME,
                                             _BATCH_ACCOUNT_KEY)

batch_client = batch.BatchServiceClient(
    credentials,
    base_url=_BATCH_ACCOUNT_URL)
```

Next, a pool of compute nodes is created in the Batch account with a call to `create_pool` .

```

def create_pool(batch_service_client, pool_id,
                resource_files, publisher, offer, sku):
    """
    Creates a pool of compute nodes with the specified OS settings.

    :param batch_service_client: A Batch service client.
    :type batch_service_client: `azure.batch.BatchServiceClient`
    :param str pool_id: An ID for the new pool.
    :param list resource_files: A collection of resource files for the pool's
    start task.
    :param str publisher: Marketplace image publisher
    :param str offer: Marketplace image offer
    :param str sku: Marketplace image sku
    """
    print('Creating pool [{}]...'.format(pool_id))

    # Create a new pool of Linux compute nodes using an Azure Virtual Machines
    # Marketplace image. For more information about creating pools of Linux
    # nodes, see:
    # https://azure.microsoft.com/documentation/articles/batch-linux-nodes/

    # Specify the commands for the pool's start task. The start task is run
    # on each node as it joins the pool, and when it's rebooted or re-imaged.
    # We use the start task to prep the node for running our task script.
    task_commands = [
        # Copy the python_tutorial_task.py script to the "shared" directory
        # that all tasks that run on the node have access to.
        'cp -r $AZ_BATCH_TASK_WORKING_DIR/* $AZ_BATCH_NODE_SHARED_DIR',
        # Install pip and the dependencies for cryptography
        'apt-get update',
        'apt-get -y install python-pip',
        'apt-get -y install build-essential libssl-dev libffi-dev python-dev',
        # Install the azure-storage module so that the task script can access
        # Azure Blob storage
        'pip install azure-storage']

    # Get the node agent SKU and image reference for the virtual machine
    # configuration.
    # For more information about the virtual machine configuration, see:
    # https://azure.microsoft.com/documentation/articles/batch-linux-nodes/
    sku_to_use, image_ref_to_use = \
        common.helpers.select_latest_verified_vm_image_with_node_agent_sku(
            batch_service_client, publisher, offer, sku)

    new_pool = batch.models.PoolAddParameter(
        id=pool_id,
        virtual_machine_configuration=batch.models.VirtualMachineConfiguration(
            image_reference=image_ref_to_use,
            node_agent_sku_id=sku_to_use),
        vm_size=_POOL_VM_SIZE,
        target_dedicated=_POOL_NODE_COUNT,
        start_task=batch.models.StartTask(
            command_line=
                common.helpers.wrap_commands_in_shell('linux', task_commands),
            run_elevated=True,
            wait_for_success=True,
            resource_files=resource_files),
    )

    try:
        batch_service_client.pool.add(new_pool)
    except batch.models.batch_error.BatchErrorException as err:
        print_batch_exception(err)
        raise

```

When you create a pool, you define a [PoolAddParameter](#) that specifies several properties for the pool:

- **ID** of the pool (*id* - required)

As with most entities in Batch, your new pool must have a unique ID within your Batch account. Your code refers to this pool using its ID, and it's how you identify the pool in the Azure [portal](#).

- **Number of compute nodes** (*target_dedicated* - required)

This property specifies how many VMs should be deployed in the pool. It is important to note that all Batch accounts have a default **quota** that limits the number of **cores** (and thus, compute nodes) in a Batch account. You can find the default quotas and instructions on how to [increase a quota](#) (such as the maximum number of cores in your Batch account) in [Quotas and limits for the Azure Batch service](#). If you find yourself asking "Why won't my pool reach more than X nodes?" this core quota may be the cause.

- **Operating system** for nodes (*virtual_machine_configuration* **or** *cloud_service_configuration* - required)

In *python_tutorial_client.py*, we create a pool of Linux nodes using a [VirtualMachineConfiguration](#). The `select_latest_verified_vm_image_with_node_agent_sku` function in `common.helpers` simplifies working with [Azure Virtual Machines Marketplace](#) images. See [Provision Linux compute nodes in Azure Batch pools](#) for more information about using Marketplace images.

- **Size of compute nodes** (*vm_size* - required)

Since we're specifying Linux nodes for our [VirtualMachineConfiguration](#), we specify a VM size (`STANDARD_A1` in this sample) from [Sizes for virtual machines in Azure](#). Again, see [Provision Linux compute nodes in Azure Batch pools](#) for more information.

- **Start task** (*start_task* - not required)

Along with the above physical node properties, you may also specify a [StartTask](#) for the pool (it is not required). The StartTask executes on each node as that node joins the pool, and each time a node is restarted. The StartTask is especially useful for preparing compute nodes for the execution of tasks, such as installing the applications that your tasks run.

In this sample application, the StartTask copies the files that it downloads from Storage (which are specified by using the StartTask's **resource_files** property) from the StartTask *working directory* to the *shared* directory that all tasks running on the node can access. Essentially, this copies `python_tutorial_task.py` to the shared directory on each node as the node joins the pool, so that any tasks that run on the node can access it.

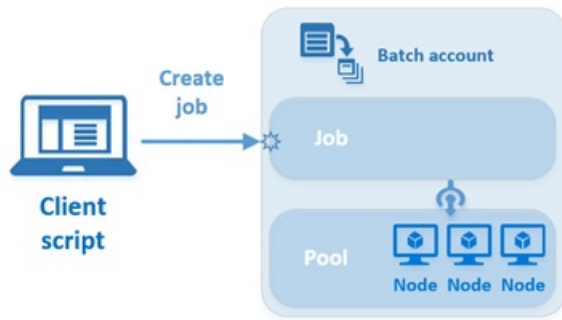
You may notice the call to the `wrap_commands_in_shell` helper function. This function takes a collection of separate commands and creates a single command line appropriate for a task's command-line property.

Also notable in the code snippet above is the use of two environment variables in the **command_line** property of the StartTask: `AZ_BATCH_TASK_WORKING_DIR` and `AZ_BATCH_NODE_SHARED_DIR`. Each compute node within a Batch pool is automatically configured with several environment variables that are specific to Batch. Any process that is executed by a task has access to these environment variables.

TIP

To find out more about the environment variables that are available on compute nodes in a Batch pool, as well as information on task working directories, see [Environment settings for tasks](#) and [Files and directories](#) in the [overview of Azure Batch features](#).

Step 4: Create Batch job



A Batch **job** is a collection of tasks, and is associated with a pool of compute nodes. The tasks in a job execute on the associated pool's compute nodes.

You can use a job not only for organizing and tracking tasks in related workloads, but also for imposing certain constraints--such as the maximum runtime for the job (and by extension, its tasks) and job priority in relation to other jobs in the Batch account. In this example, however, the job is associated only with the pool that was created in step #3. No additional properties are configured.

All Batch jobs are associated with a specific pool. This association indicates which nodes the job's tasks execute on. You specify the pool by using the [PoolInformation](#) property, as shown in the code snippet below.

```
def create_job(batch_service_client, job_id, pool_id):
    """
    Creates a job with the specified ID, associated with the specified pool.

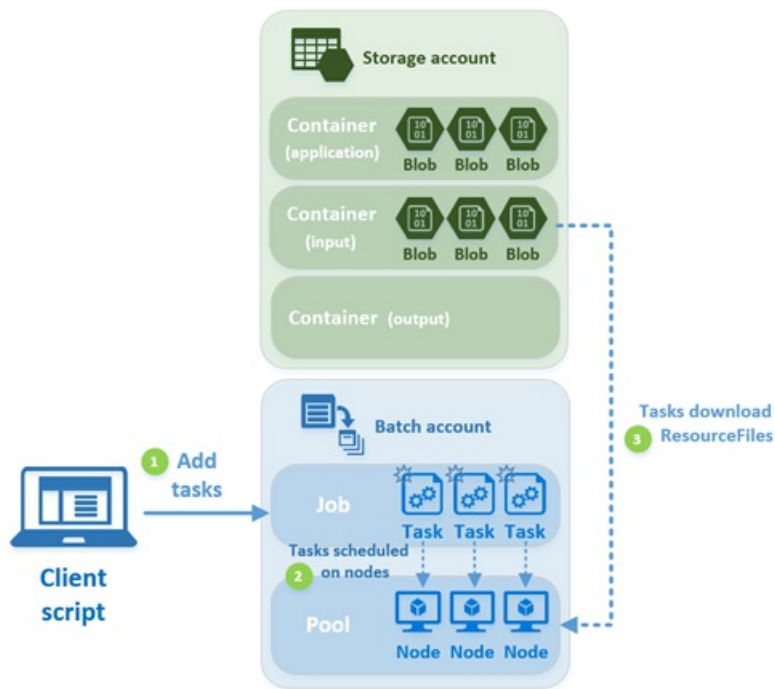
    :param batch_service_client: A Batch service client.
    :type batch_service_client: `azure.batch.BatchServiceClient`
    :param str job_id: The ID for the job.
    :param str pool_id: The ID for the pool.
    """
    print('Creating job [{}]...'.format(job_id))

    job = batch.models.JobAddParameter(
        job_id,
        batch.models.PoolInformation(pool_id=pool_id))

    try:
        batch_service_client.job.add(job)
    except batchmodels.batch_error.BatchErrorException as err:
        print_batch_exception(err)
        raise
```

Now that a job has been created, tasks are added to perform the work.

Step 5: Add tasks to job



(1) Tasks are added to the job, (2) the tasks are scheduled to run on nodes, and (3) the tasks download the data files to process

Batch **tasks** are the individual units of work that execute on the compute nodes. A task has a command line and runs the scripts or executables that you specify in that command line.

To actually perform work, tasks must be added to a job. Each [CloudTask](#) is configured with a command-line property and [ResourceFiles](#) (as with the pool's `StartTask`) that the task downloads to the node before its command line is automatically executed. In the sample, each task processes only one file. Thus, its `ResourceFiles` collection contains a single element.

```
def add_tasks(batch_service_client, job_id, input_files,
             output_container_name, output_container_sas_token):
    """
    Adds a task for each input file in the collection to the specified job.

    :param batch_service_client: A Batch service client.
    :type batch_service_client: `azure.batch.BatchServiceClient`
    :param str job_id: The ID of the job to which to add the tasks.
    :param list input_files: A collection of input files. One task will be
        created for each input file.
    :param output_container_name: The ID of an Azure Blob storage container to
        which the tasks will upload their results.
    :param output_container_sas_token: A SAS token granting write access to
        the specified Azure Blob storage container.
    """

    print('Adding {} tasks to job [{}]'...'.format(len(input_files), job_id))

    tasks = list()

    for input_file in input_files:

        command = ['python $AZ_BATCH_NODE_SHARED_DIR/python_tutorial_task.py '
                   '--filepath {} --numwords {} --storageaccount {} '
                   '--storagecontainer {} --sastoken "{}"'.format(
                       input_file.file_path,
                       '3',
                       _STORAGE_ACCOUNT_NAME,
                       output_container_name,
                       output_container_sas_token)]

        tasks.append(batch.models.TaskAddParameter(
            'topNtask{}'.format(input_files.index(input_file)),
            wrap_commands_in_shell('linux', command),
            resource_files=[input_file]
        ))

    batch_service_client.task.add_collection(job_id, tasks)
```

IMPORTANT

When they access environment variables such as `$AZ_BATCH_NODE_SHARED_DIR` or execute an application not found in the node's `PATH`, task command lines must invoke the shell explicitly, such as with `/bin/sh -c MyTaskApplication $MY_ENV_VAR`. This requirement is unnecessary if your tasks execute an application in the node's `PATH` and do not reference any environment variables.

Within the `for` loop in the code snippet above, you can see that the command line for the task is constructed with five command-line arguments that are passed to *python_tutorial_task.py*:

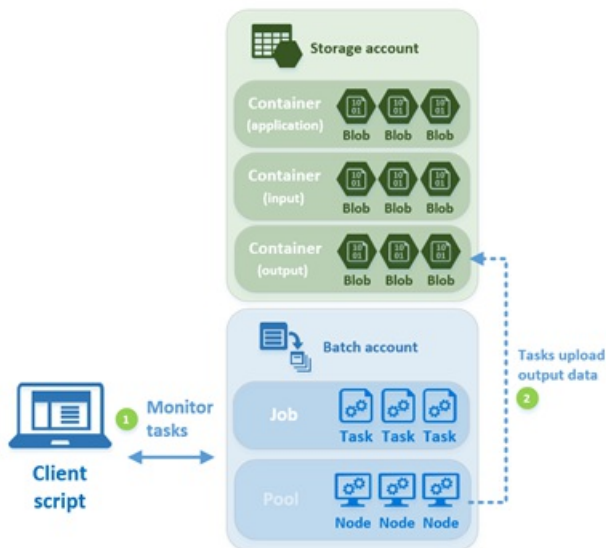
1. **filepath**: This is the local path to the file as it exists on the node. When the ResourceFile object in `upload_file_to_container` was created in Step 2 above, the file name was used for this property (the `file_path` parameter in the ResourceFile constructor). This indicates that the file can be found in the same directory on the node as *python_tutorial_task.py*.
2. **numwords**: The top *N* words should be written to the output file.
3. **storageaccount**: The name of the Storage account that owns the container to which the task output should be uploaded.
4. **storagecontainer**: The name of the Storage container to which the output files should be uploaded.
5. **sastoken**: The shared access signature (SAS) that provides write access to the **output** container in Azure Storage. The *python_tutorial_task.py* script uses this shared access signature when creates its BlockBlobService

reference. This provides write access to the container without requiring an access key for the storage account.

```
# NOTE: Taken from python_tutorial_task.py

# Create the blob client using the container's SAS token.
# This allows us to create a client that provides write
# access only to the container.
blob_client = azureblob.BlockBlobService(account_name=args.storageaccount,
                                          sas_token=args.sas_token)
```

Step 6: Monitor tasks



The script (1) monitors the tasks for completion status, and (2) the tasks upload result data to Azure Storage

When tasks are added to a job, they are automatically queued and scheduled for execution on compute nodes within the pool associated with the job. Based on the settings you specify, Batch handles all task queuing, scheduling, retrying, and other task administration duties for you.

There are many approaches to monitoring task execution. The `wait_for_tasks_to_complete` function in `python_tutorial_client.py` provides a simple example of monitoring tasks for a certain state, in this case, the `completed` state.

```
def wait_for_tasks_to_complete(batch_service_client, job_id, timeout):
    """
    Returns when all tasks in the specified job reach the Completed state.

    :param batch_service_client: A Batch service client.
    :type batch_service_client: `azure.batch.BatchServiceClient`
    :param str job_id: The id of the job whose tasks should be to monitored.
    :param timedelta timeout: The duration to wait for task completion. If all
        tasks in the specified job do not reach Completed state within this time
        period, an exception will be raised.
    """
    timeout_expiration = datetime.datetime.now() + timeout

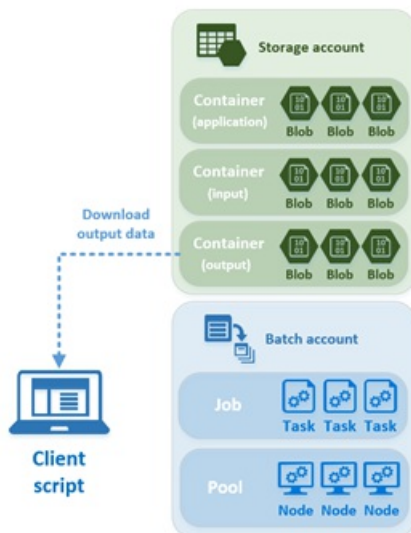
    print("Monitoring all tasks for 'Completed' state, timeout in {}..."
          .format(timeout), end='')

    while datetime.datetime.now() < timeout_expiration:
        print('.', end='')
        sys.stdout.flush()
        tasks = batch_service_client.task.list(job_id)

        incomplete_tasks = [task for task in tasks if
                             task.state != batchmodels.TaskState.completed]
        if not incomplete_tasks:
            print()
            return True
        else:
            time.sleep(1)

    print()
    raise RuntimeError("ERROR: Tasks did not reach 'Completed' state within "
                      "timeout period of " + str(timeout))
```

Step 7: Download task output



Now that the job is completed, the output from the tasks can be downloaded from Azure Storage. This is done with a call to `download_blobs_from_container` in `python_tutorial_client.py`:


```
def download_blobs_from_container(block_blob_client,
                                container_name, directory_path):
    """
    Downloads all blobs from the specified Azure Blob storage container.

    :param block_blob_client: A blob service client.
    :type block_blob_client: `azure.storage.blob.BlockBlobService`
    :param container_name: The Azure Blob storage container from which to
        download files.
    :param directory_path: The local directory to which to download the files.
    """
    print('Downloading all files from container [{}]...'.format(
        container_name))

    container_blobs = block_blob_client.list_blobs(container_name)

    for blob in container_blobs.items:
        destination_file_path = os.path.join(directory_path, blob.name)

        block_blob_client.get_blob_to_path(container_name,
                                            blob.name,
                                            destination_file_path)

        print(' Downloaded blob [{}] from container [{}] to {}'.format(
            blob.name,
            container_name,
            destination_file_path))

    print(' Download complete!')
```

NOTE

The call to `download_blobs_from_container` in *python_tutorial_client.py* specifies that the files should be downloaded to your home directory. Feel free to modify this output location.

Step 8: Delete containers

Because you are charged for data that resides in Azure Storage, it is always a good idea to remove any blobs that are no longer needed for your Batch jobs. In *python_tutorial_client.py*, this is done with three calls to [BlockBlobService.delete_container](#):

```
# Clean up storage resources
print('Deleting containers...')
blob_client.delete_container(app_container_name)
blob_client.delete_container(input_container_name)
blob_client.delete_container(output_container_name)
```

Step 9: Delete the job and the pool

In the final step, you are prompted to delete the job and the pool that were created by the *python_tutorial_client.py* script. Although you are not charged for jobs and tasks themselves, you *are* charged for compute nodes. Thus, we recommend that you allocate nodes only as needed. Deleting unused pools can be part of your maintenance process.

The BatchServiceClient's [JobOperations](#) and [PoolOperations](#) both have corresponding deletion methods, which are called if you confirm deletion:

```
# Clean up Batch resources (if the user so chooses).
if query_yes_no('Delete job?') == 'yes':
    batch_client.job.delete(_JOB_ID)

if query_yes_no('Delete pool?') == 'yes':
    batch_client.pool.delete(_POOL_ID)
```

IMPORTANT

Keep in mind that you are charged for compute resources--deleting unused pools will minimize cost. Also, be aware that deleting a pool deletes all compute nodes within that pool, and that any data on the nodes will be unrecoverable after the pool is deleted.

Run the sample script

When you run the `python_tutorial_client.py` script from the tutorial [code sample](#), the console output is similar to the following. There is a pause at `Monitoring all tasks for 'Completed' state, timeout in 0:20:00...` while the pool's compute nodes are created, started, and the commands in the pool's start task are executed. Use the [Azure portal](#) to monitor your pool, compute nodes, job, and tasks during and after execution. Use the [Azure portal](#) or the [Microsoft Azure Storage Explorer](#) to view the Storage resources (containers and blobs) that are created by the application.

TIP

Run the `python_tutorial_client.py` script from within the `azure-batch-samples/Python/Batch/article_samples` directory. It uses a relative path for the `common.helpers` module import, so you might see `ImportError: No module named 'common'` if you don't run the script from within this directory.

Typical execution time is **approximately 5-7 minutes** when you run the sample in its default configuration.

Sample start: 2016-05-20 22:47:10

```
Uploading file /home/user/py_tutorial/python_tutorial_task.py to container [application]...
Uploading file /home/user/py_tutorial/data/taskdata1.txt to container [input]...
Uploading file /home/user/py_tutorial/data/taskdata2.txt to container [input]...
Uploading file /home/user/py_tutorial/data/taskdata3.txt to container [input]...
Creating pool [PythonTutorialPool]...
Creating job [PythonTutorialJob]...
Adding 3 tasks to job [PythonTutorialJob]...
Monitoring all tasks for 'Completed' state, timeout in
0:20:00.....
    Success! All tasks reached the 'Completed' state within the specified timeout period.
Downloading all files from container [output]...
    Downloaded blob [taskdata1_OUTPUT.txt] from container [output] to /home/user/taskdata1_OUTPUT.txt
    Downloaded blob [taskdata2_OUTPUT.txt] from container [output] to /home/user/taskdata2_OUTPUT.txt
    Downloaded blob [taskdata3_OUTPUT.txt] from container [output] to /home/user/taskdata3_OUTPUT.txt
    Download complete!
Deleting containers...
```

Sample end: 2016-05-20 22:53:12
Elapsed time: 0:06:02

Delete job? [Y/n]
Delete pool? [Y/n]

Press ENTER to exit...

Next steps

Feel free to make changes to *python_tutorial_client.py* and *python_tutorial_task.py* to experiment with different compute scenarios. For example, try adding an execution delay to *python_tutorial_task.py* to simulate long-running tasks and monitor them in the portal. Try adding more tasks or adjusting the number of compute nodes. Add logic to check for and allow the use of an existing pool to speed execution time.

Now that you're familiar with the basic workflow of a Batch solution, it's time to dig in to the additional features of the Batch service.

- Review the [Overview of Azure Batch features](#) article, which we recommend if you're new to the service.
- Start on the other Batch development articles under **Development in-depth** in the [Batch learning path](#).
- Check out a different implementation of processing the "top N words" workload with Batch in the [TopNWords](#) sample.

Deploy applications to compute nodes with Batch application packages

2/27/2017 • 14 min to read • [Edit Online](#)

The application packages feature of Azure Batch provides easy management of task applications and their deployment to the compute nodes in your pool. With application packages, you can upload and manage multiple versions of the applications your tasks run, including their supporting files. You can then automatically deploy one or more of these applications to the compute nodes in your pool.

In this article, you will learn how to upload and manage application packages in the Azure portal. You will then learn how to install them on a pool's compute nodes with the [Batch .NET](#) library.

NOTE

The application packages feature described here supersedes the "Batch Apps" feature available in previous versions of the service.

Application package requirements

You must [link an Azure Storage account](#) to your Batch account to use application packages.

The application packages feature discussed in this article is compatible *only* with Batch pools that were created after 10 March 2016. Application packages will not be deployed to compute nodes in pools created before this date.

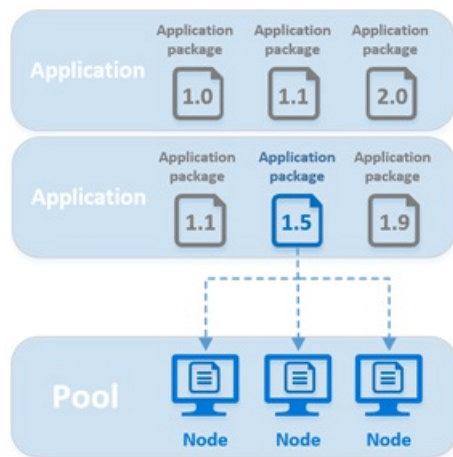
This feature was introduced in [Batch REST API](#) version 2015-12-01.2.2 and the corresponding [Batch .NET](#) library version 3.1.0. We recommend that you always use the latest API version when working with Batch.

IMPORTANT

Currently, only *CloudServiceConfiguration* pools support application packages. You cannot use Application packages in pools created by using *VirtualMachineConfiguration* images. See the [Virtual machine configuration](#) section of [Provision Linux compute nodes in Azure Batch pools](#) for more information about the two different configurations.

About applications and application packages

Within Azure Batch, an *application* refers to a set of versioned binaries that can be automatically downloaded to the compute nodes in your pool. An *application package* refers to a *specific set* of those binaries and represents a given *version* of the application.



Applications

An application in Batch contains one or more application packages and specifies configuration options for the application. For example, an application can specify the default application package version to install on compute nodes and whether its packages can be updated or deleted.

Application packages

An application package is a .zip file that contains the application binaries and supporting files that are required for execution by your tasks. Each application package represents a specific version of the application.

You can specify application packages at the pool and task level. You can specify one or more of these packages and (optionally) a version when you create a pool or task.

- **Pool application packages** are deployed to *every* node in the pool. Applications are deployed when a node joins a pool, and when it is rebooted or reimaged.

Pool application packages are appropriate when all nodes in a pool execute a job's tasks. You can specify one or more application packages when you create a pool, and you can add or update an existing pool's packages. If you update an existing pool's application packages, you must restart its nodes to install the new package.

- **Task application packages** are deployed only to a compute node scheduled to run a task, just before running the task's command line. If the specified application package and version is already on the node, it is not redeployed and the existing package is used.

Task application packages are useful in shared-pool environments, where different jobs are run on one pool, and the pool is not deleted when a job is completed. If your job has less tasks than nodes in the pool, task application packages can minimize data transfer since your application is deployed only to the nodes that run tasks.

Other scenarios that can benefit from task application packages are jobs that use a particularly large application, but for only a small number of tasks. For example, a pre-processing stage or a merge task, where the pre-processing or merge application is heavyweight.

IMPORTANT

There are restrictions on the number of applications and application packages within a Batch account, as well as the maximum application package size. See [Quotas and limits for the Azure Batch service](#) for details about these limits.

Benefits of application packages

Application packages can simplify the code in your Batch solution and lower the overhead required to manage the applications that your tasks run.

Your pool's start task doesn't have to specify a long list of individual resource files to install on the nodes. You don't have to manually manage multiple versions of your application files in Azure Storage, or on your nodes. And, you don't need to worry about generating [SAS URLs](#) to provide access to the files in your Storage account. Batch works in the background with Azure Storage to store application packages and deploy them to compute nodes.

Upload and manage applications

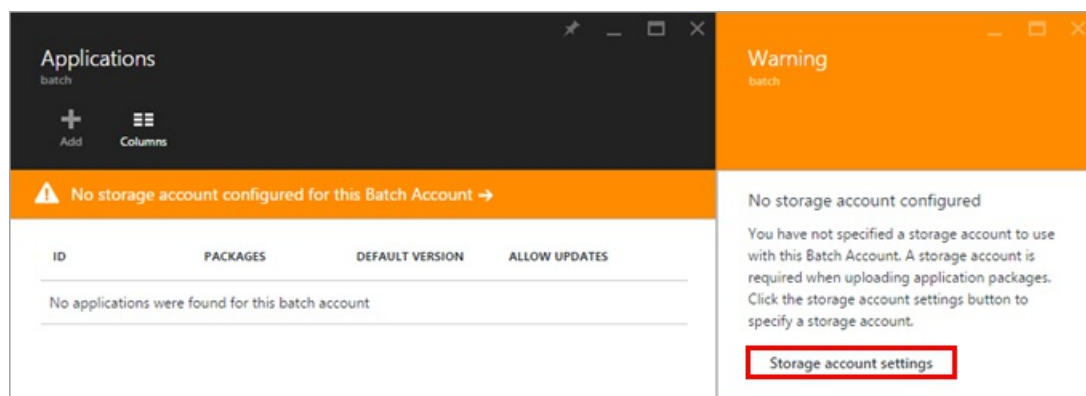
You can use the [Azure portal](#) or the [Batch Management .NET](#) library to manage the application packages in your Batch account. In the next few sections, we first link a Storage account, then discuss adding applications and packages and managing them with the portal.

Link a Storage account

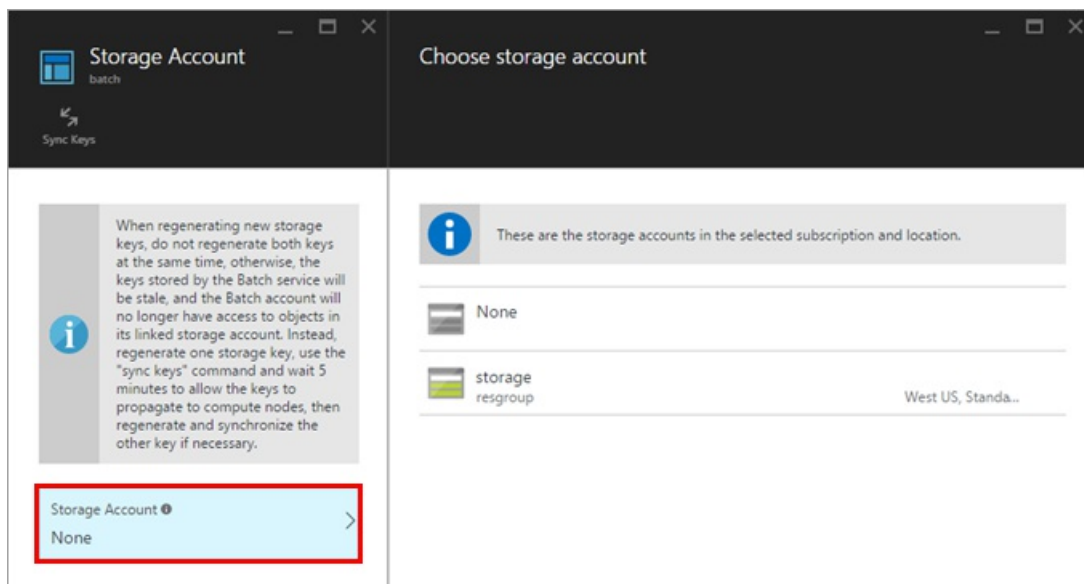
To use application packages, you must first link an Azure Storage account to your Batch account. If you have not yet configured a Storage account for your Batch account, the Azure portal will display a warning the first time you click the **Applications** tile in the **Batch account** blade.

IMPORTANT

Batch currently supports *only* the **General purpose** storage account type as described in step 5, [Create a storage account](#), in [About Azure storage accounts](#). When you link an Azure Storage account to your Batch account, link *only* a **General purpose** storage account.



The Batch service uses the associated Storage account for the storage and retrieval of application packages. After you've linked the two accounts, Batch can automatically deploy the packages stored in the linked Storage account to your compute nodes. Click **Storage account settings** on the **Warning** blade, and then click **Storage Account** on the **Storage Account** blade to link a storage account to your Batch account.



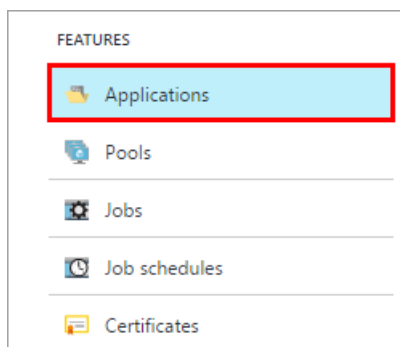
We recommend that you create a storage account *specifically* for use with your Batch account, and select it here. For details about how to create a storage account, see "Create a storage account" in [About Azure storage accounts](#). After you've created a Storage account, you can then link it to your Batch account by using the **Storage Account** blade.

WARNING

Because Batch uses Azure Storage to store your application packages, you are **charged as normal** for the block blob data. Be sure to consider the size and number of your application packages, and periodically remove deprecated packages to minimize cost.

View current applications

To view the applications in your Batch account, click the **Applications** menu item in the left menu while viewing the **Batch account** blade.



This opens the **Applications** blade:

ID	PACKAGES	DEFAULT VERSION	ALLOW UPDATES	
blender	2	2.76b	Yes	...
litware	1		No	...

The **Applications** blade displays the ID of each application in your account and the following properties:

- **Packages**--The number of versions associated with this application.
- **Default version**--The version that will be installed if you do not specify a version when you set the application for a pool. This setting is optional.
- **Allow updates**--The value that specifies whether package updates, deletions, and additions are allowed. If this is set to **No**, package updates and deletions are disabled for the application. Only new application package versions can be added. The default is **Yes**.

View application details

Click an application in the **Applications** blade to open the blade that includes the details for that application.

blender
Application

Save Discard

Packages
2

Allow updates ⓘ

Default version ⓘ
2.76b ▼

Display name ⓘ
Blender

In the application details blade, you can configure the following settings for your application.

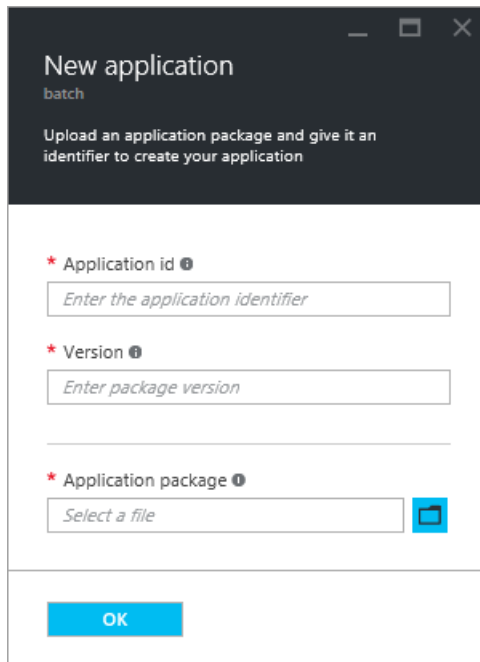
- **Allow updates**--Specify whether its application packages can be updated or deleted. See "Update or Delete an application package" later in this article.
- **Default version**--Specify a default application package to deploy to compute nodes.
- **Display name**--Specify a "friendly" name that your Batch solution can use when it displays information

about the application, such as in the UI of a service that you provide your customers through Batch.

Add a new application

To create a new application, add an application package and specify a new, unique application ID. The first application package that you add with the new application ID will also create the new application.

Click **Add** on the **Applications** blade to open the **New application** blade.



The **New application** blade provides the following fields to specify the settings of your new application and application package.

Application id

This field specifies the ID of your new application, which is subject to the standard Azure Batch ID validation rules:

- Can contain any combination of alphanumeric characters, including hyphens and underscores.
- Cannot contain more than 64 characters.
- Must be unique within the Batch account.
- Is case preserving and case insensitive.

Version

Specifies the version of the application package you are uploading. Version strings are subject to the following validation rules:

- Can contain any combination of alphanumeric characters, including hyphens, underscores, and periods.
- Cannot contain more than 64 characters.
- Must be unique within the application.
- Case preserving, and case insensitive.

Application package

This field specifies the .zip file that contains the application binaries and supporting files that are required to execute the application. Click the **Select a file** box or the folder icon to browse to and select a .zip file that contains your application's files.

After you've selected a file, click **OK** to begin the upload to Azure Storage. When the upload operation is complete, you will be notified and the blade will close. Depending on the size of the file that you are uploading

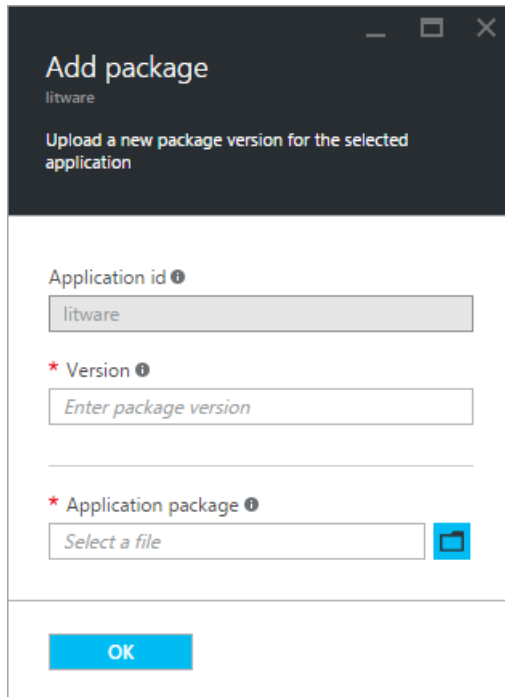
and the speed of your network connection, this operation may take some time.

WARNING

Do not close the **New application** blade before the upload operation is complete. Doing so will stop the upload process.

Add a new application package

To add a new application package version for an existing application, select an application in the **Applications** blade, click **Packages**, then click **Add** to open the **Add package** blade.

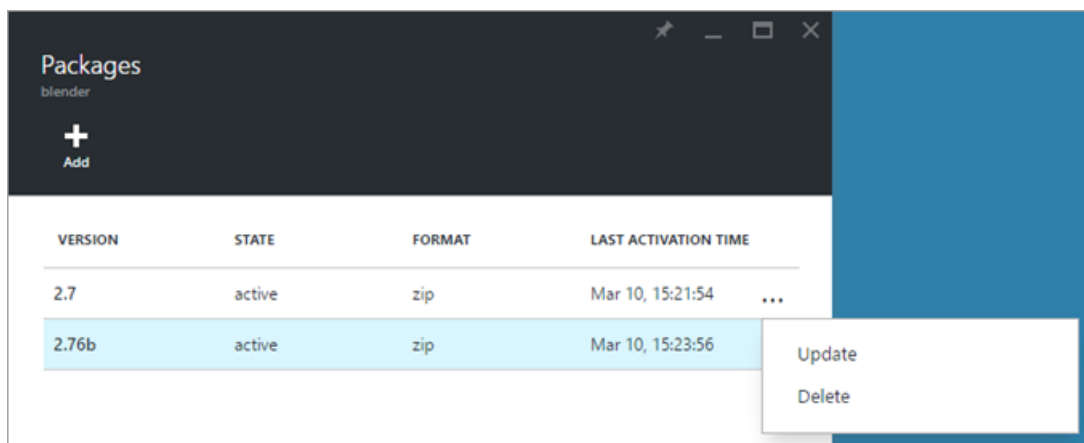


The screenshot shows a dialog box titled "Add package" for the application "litware". The subtitle is "Upload a new package version for the selected application". The dialog contains three input fields: "Application id" (disabled, showing "litware"), "Version" (required, with a red asterisk, showing "Enter package version"), and "Application package" (required, with a red asterisk, showing "Select a file" and a file selection icon). An "OK" button is at the bottom.

As you can see, the fields match those of the **New application** blade, but the **Application id** box is disabled. As you did for the new application, specify the **Version** for your new package, browse to your **Application package** .zip file, then click **OK** to upload the package.

Update or delete an application package

To update or delete an existing application package, open the details blade for the application, click **Packages** to open the **Packages** blade, click the **ellipsis** in the row of the application package that you want to modify, and select the action that you want to perform.



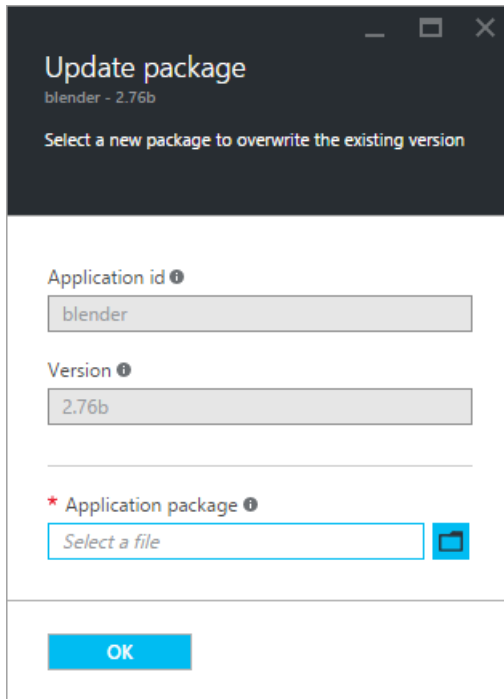
The screenshot shows the "Packages" blade for the application "blender". It features a table with columns: VERSION, STATE, FORMAT, and LAST ACTIVATION TIME. There are two rows of packages. The second row is selected, and a context menu is open showing "Update" and "Delete" options.

VERSION	STATE	FORMAT	LAST ACTIVATION TIME
2.7	active	zip	Mar 10, 15:21:54
2.76b	active	zip	Mar 10, 15:23:56

Update

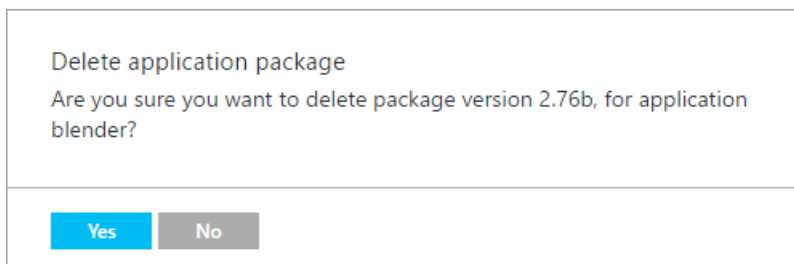
When you click **Update**, the *Update package* blade is displayed. This blade is similar to the *New application*

package blade, however only the package selection field is enabled, allowing you to specify a new ZIP file to upload.



Delete

When you click **Delete**, you are asked to confirm the deletion of the package version, and Batch deletes the package from Azure Storage. If you delete the default version of an application, the **Default version** setting is removed for the application.



Install applications on compute nodes

Now that you've seen how to manage application packages with the Azure portal, we can discuss how to deploy them to compute nodes and run them with Batch tasks.

Install pool application packages

To install an application package on all compute nodes in a pool, specify one or more application package *references* for the pool. The application packages that you specify for a pool are installed on each compute node when that node joins the pool, and when the node is rebooted or reimaged.

In Batch .NET, specify one or more [CloudPool.ApplicationPackageReferences](#) when you create a new pool, or for an existing pool. The [ApplicationPackageReference](#) class specifies an application ID and version to install on a pool's compute nodes.

```
// Create the unbound CloudPool
CloudPool myCloudPool =
    batchClient.PoolOperations.CreatePool(
        poolId: "myPool",
        targetDedicated: "1",
        virtualMachineSize: "small",
        cloudServiceConfiguration: new CloudServiceConfiguration(osFamily: "4"));

// Specify the application and version to install on the compute nodes
myCloudPool.ApplicationPackageReferences = new List<ApplicationPackageReference>
{
    new ApplicationPackageReference {
        ApplicationId = "litware",
        Version = "1.1001.2b" }
};

// Commit the pool so that it's created in the Batch service. As the nodes join
// the pool, the specified application package will be installed on each.
await myCloudPool.CommitAsync();
```

IMPORTANT

If an application package deployment fails for any reason, the Batch service marks the node **unusable**, and no tasks will be scheduled for execution on that node. In this case, you should **restart** the node to reinitiate the package deployment. Restarting the node will also enable task scheduling again on the node.

Install task application packages

Similar to a pool, you specify application package *references* for a task. When a task is scheduled to run on a node, the package is downloaded and extracted just before the task's command line is executed. If a specified package and version is already installed on the node, the package is not downloaded and the existing package is used.

To install a task application package, configure the task's `CloudTask.ApplicationPackageReferences` property:

```
CloudTask task =
    new CloudTask(
        "litwaretask001",
        "cmd /c %AZ_BATCH_APP_PACKAGE_LITWARE%\litware.exe -args -here");

task.ApplicationPackageReferences = new List<ApplicationPackageReference>
{
    new ApplicationPackageReference
    {
        ApplicationId = "litware",
        Version = "1.1001.2b"
    }
};
```

Execute the installed applications

The packages that you've specified for a pool or task are downloaded and extracted to a named directory within the `AZ_BATCH_ROOT_DIR` of the node. Batch also creates an environment variable that contains the path to the named directory. Your task command lines use this environment variable when referencing the application on the node. The variable is in the following format:

```
AZ_BATCH_APP_PACKAGE_APPLICATIONID#version
```

`APPLICATIONID` and `version` are values that correspond to the application and package version you've

specified for deployment. For example, if you specified that version 2.7 of application *blender* should be installed, your task command lines would use this environment variable to access its files:

```
AZ_BATCH_APP_PACKAGE_BLENDER#2.7
```

When you upload an application package, you can specify a default version to deploy to your compute nodes. If you have specified a default version for an application, you can omit the version suffix when you reference the application. You can specify the default application version in the Azure portal, on the Applications blade, as shown in [Upload and manage applications](#).

For example, if you set "2.7" as the default version for application *blender*, your tasks can reference the following environment variable and they will execute version 2.7:

```
AZ_BATCH_APP_PACKAGE_BLENDER
```

The following code snippet shows an example task command line that launches the default version of the *blender* application:

```
string taskId = "blendertask01";
string commandLine =
    @"cmd /c %AZ_BATCH_APP_PACKAGE_BLENDER%\blender.exe -args -here";
CloudTask blenderTask = new CloudTask(taskId, commandLine);
```

TIP

See [Environment settings for tasks](#) in the [Batch feature overview](#) for more information about compute node environment settings.

Update a pool's application packages

If an existing pool has already been configured with an application package, you can specify a new package for the pool. If you specify a new package reference for a pool, the following apply:

- All new nodes that join the pool and any existing node that is rebooted or reimaged will install the newly specified package.
- Compute nodes that are already in the pool when you update the package references do not automatically install the new application package. These compute nodes must be rebooted or reimaged to receive the new package.
- When a new package is deployed, the created environment variables reflect the new application package references.

In this example, the existing pool has version 2.7 of the *blender* application configured as one of its [CloudPool.ApplicationPackageReferences](#). To update the pool's nodes with version 2.76b, specify a new [ApplicationPackageReference](#) with the new version, and commit the change.

```
string newVersion = "2.76b";
CloudPool boundPool = await batchClient.PoolOperations.GetPoolAsync("myPool");
boundPool.ApplicationPackageReferences = new List<ApplicationPackageReference>
{
    new ApplicationPackageReference {
        ApplicationId = "blender",
        Version = newVersion }
};
await boundPool.CommitAsync();
```

Now that the new version has been configured, any *new* node that joins the pool will have version 2.76b

deployed to it. To install 2.76b on the nodes that are *already* in the pool, reboot or reimagine them. Note that rebooted nodes will retain the files from previous package deployments.

List the applications in a Batch account

You can list the applications and their packages in a Batch account by using the [ApplicationOperations.ListApplicationSummaries](#) method.

```
// List the applications and their application packages in the Batch account.
List<ApplicationSummary> applications = await
batchClient.ApplicationOperations.ListApplicationSummaries().ToListAsync();
foreach (ApplicationSummary app in applications)
{
    Console.WriteLine("ID: {0} | Display Name: {1}", app.Id, app.DisplayName);

    foreach (string version in app.Versions)
    {
        Console.WriteLine("  {0}", version);
    }
}
```

Wrap up

With application packages, you can help your customers select the applications for their jobs and specify the exact version to use when processing jobs with your Batch-enabled service. You might also provide the ability for your customers to upload and track their own applications in your service.

Next steps

- The [Batch REST API](#) also provides support to work with application packages. For example, see the [applicationPackageReferences](#) element in [Add a pool to an account](#) for information about how to specify packages to install by using the REST API. See [Applications](#) for details about how to obtain application information by using the Batch REST API.
- Learn how to programmatically [manage Azure Batch accounts and quotas with Batch Management .NET](#). The [Batch Management .NET](#) library can enable account creation and deletion features for your Batch application or service.

Create an automatic scaling formula for scaling compute nodes in a Batch pool

3/15/2017 • 27 min to read • [Edit Online](#)

With automatic scaling, the Azure Batch service can dynamically add or remove compute nodes in a pool based on parameters that you define. You can potentially save both time and money by automatically adjusting the amount of compute power used by your application--add nodes as your job's task demands increase, and remove them when they decrease.

You enable automatic scaling on a pool of compute nodes by associating with it an *autoscale formula* that you define, such as with the [PoolOperations.EnableAutoScale](#) method in the [Batch .NET](#) library. The Batch service then uses this formula to determine the number of compute nodes that are needed to execute your workload. Batch responds to service metrics data samples that are collected periodically, and adjusts the number of compute nodes in the pool at a configurable interval based on your formula.

You can enable automatic scaling when a pool is created, or on an existing pool. You can also change an existing formula on a pool that is "autoscale" enabled. Batch provides the ability to evaluate your formulas before assigning them to pools, as well as monitor the status of automatic scaling runs.

Automatic scaling formulas

An automatic scaling formula is a string value that you define that contains one or more statements, and is assigned to a pool's [autoScaleFormula](#) element (Batch REST) or [CloudPool.AutoScaleFormula](#) property (Batch .NET). When assigned to a pool, the Batch service uses your formula to determine the target number of compute nodes in the pool for the next interval of processing (more on intervals later). The formula string cannot exceed 8 KB in size, can include up to 100 statements that are separated by semicolons, and can include line breaks and comments.

You can think of automatic scaling formulas as using a Batch autoscale "language." Formula statements are free-formed expressions that can include both service-defined variables (variables defined by the Batch service) and user-defined variables (variables that you define). They can perform various operations on these values by using built-in types, operators, and functions. For example, a statement might take the following form:

```
$myNewVariable = function($ServiceDefinedVariable, $myCustomVariable);
```

Formulas generally contain multiple statements that perform operations on values that are obtained in previous statements. For example, first we obtain a value for `variable1`, then pass it to a function to populate `variable2`:

```
$variable1 = function1($ServiceDefinedVariable);  
$variable2 = function2($OtherServiceDefinedVariable, $variable1);
```

With these statements in your formula, your goal is to arrive at a number of compute nodes that the pool should be scaled to--the **target** number of **dedicated nodes**. This number may be higher, lower, or the same as the current number of nodes in the pool. Batch evaluates a pool's autoscale formula at a specific interval ([automatic scaling intervals](#) are discussed below). Then it will adjust the target number of nodes in the pool to the number that your autoscale formula specifies at the time of evaluation.

As a quick example, this two-line autoscale formula specifies that the number of nodes should be adjusted according to the number of active tasks, up to a maximum of 10 compute nodes:

```
$averageActiveTaskCount = avg($ActiveTasks.GetSample(TimeInterval_Minute * 15));  
$TargetDedicated = min(10, $averageActiveTaskCount);
```

The next few sections of this article discuss the various entities that will make up your autoscale formulas, including variables, operators, operations, and functions. You'll find out how to obtain various compute resource and task metrics within Batch. You can use these metrics to intelligently adjust your pool's node count based on resource usage and task status. You'll then learn how to construct a formula and enable automatic scaling on a pool by using both the Batch REST and .NET APIs. We'll finish up with a few example formulas.

IMPORTANT

Each Azure Batch account is limited to a maximum number of cores (and therefore compute nodes) that can be used for processing. The Batch service will create nodes only up to that core limit. Therefore, it may not reach the target number of compute nodes that is specified by a formula. See [Quotas and limits for the Azure Batch service](#) for information on viewing and increasing your account quotas.

Variables

You can use both **service-defined** and **user-defined** variables in your autoscale formulas. The service-defined variables are built in to the Batch service--some are read-write, and some are read-only. User-defined variables are variables that *you* define. In the two-line example formula above, `$TargetDedicated` is a service-defined variable, while `$averageActiveTaskCount` is a user-defined variable.

The tables below show both read-write and read-only variables that are defined by the Batch service.

You can **get** and **set** the values of these service-defined variables to manage the number of compute nodes in a pool:

READ-WRITE SERVICE-DEFINED VARIABLES	DESCRIPTION
<code>\$TargetDedicated</code>	The target number of dedicated compute nodes for the pool. This is the number of compute nodes that the pool should be scaled to. It is a "target" number since it's possible for a pool not to reach the target number of nodes. This can occur if the target number of nodes is modified again by a subsequent autoscale evaluation before the pool has reached the initial target. It can also happen if a Batch account node or core quota is reached before the target number of nodes is reached.
<code>\$NodeDeallocationOption</code>	The action that occurs when compute nodes are removed from a pool. Possible values are: <ul style="list-style-type: none">• requeue--Terminates tasks immediately and puts them back on the job queue so that they are rescheduled.• terminate--Terminates tasks immediately and removes them from the job queue.• taskcompletion--Waits for currently running tasks to finish and then removes the node from the pool.• retaineddata--Waits for all the local task-retained data on the node to be cleaned up before removing the node from the pool.

You can **get** the value of these service-defined variables to make adjustments that are based on metrics from the Batch service:

READ-ONLY SERVICE-DEFINED VARIABLES	DESCRIPTION
\$CPUPercent	The average percentage of CPU usage.
\$WallClockSeconds	The number of seconds consumed.
\$MemoryBytes	The average number of megabytes used.
\$DiskBytes	The average number of gigabytes used on the local disks.
\$DiskReadBytes	The number of bytes read.
\$DiskWriteBytes	The number of bytes written.
\$DiskReadOps	The count of read disk operations performed.
\$DiskWriteOps	The count of write disk operations performed.
\$NetworkInBytes	The number of inbound bytes.
\$NetworkOutBytes	The number of outbound bytes.
\$SampleNodeCount	The count of compute nodes.
\$ActiveTasks	The number of tasks in an active state.
\$RunningTasks	The number of tasks in a running state.
\$PendingTasks	The sum of \$ActiveTasks and \$RunningTasks.
\$SucceededTasks	The number of tasks that finished successfully.
\$FailedTasks	The number of tasks that failed.
\$CurrentDedicated	The current number of dedicated compute nodes.

TIP

The read-only, service-defined variables that are shown above are *objects* that provide various methods to access data associated with each. See [Obtain sample data](#) below for more information.

Types

These **types** are supported in a formula.

- double
- doubleVec
- doubleVecList
- string
- timestamp--timestamp is a compound structure that contains the following members:
 - year

- month (1-12)
- day (1-31)
- weekday (in the format of number, e.g. 1 for Monday)
- hour (in 24-hour number format, e.g. 13 means 1 PM)
- minute (00-59)
- second (00-59)
- **timeinterval**
 - TimeInterval_Zero
 - TimeInterval_100ns
 - TimeInterval_Microsecond
 - TimeInterval_Millisecond
 - TimeInterval_Second
 - TimeInterval_Minute
 - TimeInterval_Hour
 - TimeInterval_Day
 - TimeInterval_Week
 - TimeInterval_Year

Operations

These **operations** are allowed on the types that are listed above.

OPERATION	SUPPORTED OPERATORS	RESULT TYPE
double <i>operator</i> double	+, -, *, /	double
double <i>operator</i> timeinterval	*	timeinterval
doubleVec <i>operator</i> double	+, -, *, /	doubleVec
doubleVec <i>operator</i> doubleVec	+, -, *, /	doubleVec
timeinterval <i>operator</i> double	*, /	timeinterval
timeinterval <i>operator</i> timeinterval	+, -	timeinterval
timeinterval <i>operator</i> timestamp	+	timestamp
timestamp <i>operator</i> timeinterval	+	timestamp
timestamp <i>operator</i> timestamp	-	timeinterval
<i>operator</i> double	-, !	double
<i>operator</i> timeinterval	-	timeinterval
double <i>operator</i> double	<, <=, ==, >=, >, !=	double
string <i>operator</i> string	<, <=, ==, >=, >, !=	double

OPERATION	SUPPORTED OPERATORS	RESULT TYPE
timestamp <i>operator</i> timestamp	<, <=, ==, >=, >, !=	double
timeinterval <i>operator</i> timeinterval	<, <=, ==, >=, >, !=	double
double <i>operator</i> double	&&,	double

When testing a double with a ternary operator (`double ? statement1 : statement2`), nonzero is **true**, and zero is **false**.

Functions

These predefined **functions** are available for you to use in defining an automatic scaling formula.

FUNCTION	RETURN TYPE	DESCRIPTION
avg(doubleVecList)	double	Returns the average value for all values in the doubleVecList.
len(doubleVecList)	double	Returns the length of the vector that is created from the doubleVecList.
lg(double)	double	Returns the log base 2 of the double.
lg(doubleVecList)	doubleVec	Returns the componentwise log base 2 of the doubleVecList. A vec(double) must explicitly be passed for the parameter. Otherwise, the double lg(double) version is assumed.
ln(double)	double	Returns the natural log of the double.
ln(doubleVecList)	doubleVec	Returns the componentwise log base 2 of the doubleVecList. A vec(double) must explicitly be passed for the parameter. Otherwise, the double lg(double) version is assumed.
log(double)	double	Returns the log base 10 of the double.
log(doubleVecList)	doubleVec	Returns the componentwise log base 10 of the doubleVecList. A vec(double) must explicitly be passed for the single double parameter. Otherwise, the double log(double) version is assumed.
max(doubleVecList)	double	Returns the maximum value in the doubleVecList.
min(doubleVecList)	double	Returns the minimum value in the doubleVecList.
norm(doubleVecList)	double	Returns the two-norm of the vector that is created from the doubleVecList.

FUNCTION	RETURN TYPE	DESCRIPTION
percentile(doubleVec v, double p)	double	Returns the percentile element of the vector v.
rand()	double	Returns a random value between 0.0 and 1.0.
range(doubleVecList)	double	Returns the difference between the min and max values in the doubleVecList.
std(doubleVecList)	double	Returns the sample standard deviation of the values in the doubleVecList.
stop()		Stops evaluation of the autoscaling expression.
sum(doubleVecList)	double	Returns the sum of all the components of the doubleVecList.
time(string dateTime="")	timestamp	Returns the time stamp of the current time if no parameters are passed, or the time stamp of the dateTime string if it is passed. Supported dateTime formats are W3C-DTF and RFC 1123.
val(doubleVec v, double i)	double	Returns the value of the element that is at location i in vector v, with a starting index of zero.

Some of the functions that are described in the table above can accept a list as an argument. The comma-separated list is any combination of *double* and *doubleVec*. For example:

```
doubleVecList := ( (double | doubleVec)+ (, (double | doubleVec) )* )?
```

The *doubleVecList* value is converted to a single *doubleVec* prior to evaluation. For example, if `v = [1,2,3]`, then calling `avg(v)` is equivalent to calling `avg(1,2,3)`. Calling `avg(v, 7)` is equivalent to calling `avg(1,2,3,7)`.

Obtain sample data

Autoscale formulas act on metrics data (samples) that is provided by the Batch service. A formula grows or shrinks pool size based on the values that it obtains from the service. The service-defined variables that are described above are objects that provide various methods to access data that is associated with that object. For example, the following expression shows a request to get the last five minutes of CPU usage:

```
$CPUPercent.GetSample(TimeInterval_Minute * 5)
```

METHOD	DESCRIPTION
--------	-------------

METHOD	DESCRIPTION
GetSample()	<p>The <code>GetSample()</code> method returns a vector of data samples.</p> <p>A sample is 30 seconds worth of metrics data. In other words, samples are obtained every 30 seconds. But as noted below, there is a delay between when a sample is collected and when it is available to a formula. As such, not all samples for a given time period may be available for evaluation by a formula.</p> <ul style="list-style-type: none"> <code>doubleVec GetSample(double count)</code> Specifies the number of samples to obtain from the most recent samples that were collected. <p><code>GetSample(1)</code> returns the last available sample. For metrics like <code>\$CPUPercent</code>, however, this should not be used because it is impossible to know <i>when</i> the sample was collected. It might be recent, or, because of system issues, it might be much older. It is better in such cases to use a time interval as shown below.</p> <pre>doubleVec GetSample((timestamp or timeinterval) startTime [, double samplePercent])</pre> <ul style="list-style-type: none"> Specifies a time frame for gathering sample data. Optionally, it also specifies the percentage of samples that must be available in the requested time frame. <pre>\$CPUPercent.GetSample(TimeInterval_Minute * 10)</pre> <p>would return 20 samples if all samples for the last ten minutes are present in the CPUPercent history. If the last minute of history was not available, however, only 18 samples would be returned. In this case:</p> <pre>\$CPUPercent.GetSample(TimeInterval_Minute * 10, 95)</pre> <p>would fail because only 90 percent of the samples are available.</p> <pre>\$CPUPercent.GetSample(TimeInterval_Minute * 10, 80)</pre> <p>would succeed.</p> <pre>doubleVec GetSample((timestamp or timeinterval) startTime, (timestamp or timeinterval) endTime [, double samplePercent])</pre> <ul style="list-style-type: none"> Specifies a time frame for gathering data, with both a start time and an end time. <p>As mentioned above, there is a delay between when a sample is collected and when it is available to a formula. This must be considered when you use the <code>GetSample</code> method. See <code>GetSamplePercent</code> below.</p>
GetSamplePeriod()	Returns the period of samples that were taken in a historical sample data set.
Count()	Returns the total number of samples in the metric history.
HistoryBeginTime()	Returns the time stamp of the oldest available data sample for the metric.

METHOD	DESCRIPTION
GetSamplePercent()	<p>Returns the percentage of samples that are available for a given time interval. For example:</p> <pre>doubleVec GetSamplePercent((timestamp or timeinterval) startTime [, (timestamp or timeinterval) endTime])</pre> <p>Because the <code>GetSample</code> method fails if the percentage of samples returned is less than the <code>samplePercent</code> specified, you can use the <code>GetSamplePercent</code> method to check first. Then you can perform an alternate action if insufficient samples are present, without halting the automatic scaling evaluation.</p>

Samples, sample percentage, and the `GetSample()` method

The core operation of an autoscale formula is to obtain task and resource metric data and then adjust pool size based on that data. As such, it is important to have a clear understanding of how autoscale formulas interact with metrics data, or "samples."

Samples

The Batch service periodically takes *samples* of task and resource metrics and makes them available to your autoscale formulas. These samples are recorded every 30 seconds by the Batch service. However, there is typically some latency that causes a delay between when those samples were recorded and when they are made available to (and can be read by) your autoscale formulas. Additionally, due to various factors such as network or other infrastructure issues, samples may not have been recorded for a particular interval. This results in "missing" samples.

Sample percentage

When `samplePercent` is passed to the `GetSample()` method or the `GetSamplePercent()` method is called, "percent" refers to a comparison between the total *possible* number of samples that are recorded by the Batch service and the number of samples that are actually *available* to your autoscale formula.

Let's look at a 10-minute timespan as an example. Because samples are recorded every 30 seconds, within a 10 minute timespan, the maximum total number of samples that are recorded by Batch would be 20 samples (2 per minute). However, due to the inherent latency of the reporting mechanism or some other issue within the Azure infrastructure, there may be only 15 samples that are available to your autoscale formula for reading. This means that, for that 10-minute period, only **75 percent** of the total number of samples recorded are actually available to your formula.

`GetSample()` and sample ranges

Your autoscale formulas are going to be growing and shrinking your pools--adding nodes or removing nodes. Because nodes cost you money, you want to ensure that your formulas use an intelligent method of analysis that is based on sufficient data. Therefore, we recommend that you use a trending-type analysis in your formulas. This type will grow and shrink your pools based on a *range* of collected samples.

To do so, use `GetSample(interval look-back start, interval look-back end)` to return a **vector** of samples:

```
$runningTasksSample = $RunningTasks.GetSample(1 * TimeInterval_Minute, 6 * TimeInterval_Minute);
```

When the above line is evaluated by Batch, it will return a range of samples as a vector of values. For example:

```
$runningTasksSample=[1,1,1,1,1,1,1,1,1,1];
```

Once you've collected the vector of samples, you can then use functions like `min()`, `max()`, and `avg()` to derive meaningful values from the collected range.

For additional security, you can force a formula evaluation to *fail* if less than a certain sample percentage is available for a particular time period. When you force a formula evaluation to fail, you instruct Batch to cease further evaluation of the formula if the specified percentage of samples is not available--and no change to pool size will be made. To specify a required percentage of samples for the evaluation to succeed, specify it as the third parameter to `GetSample()`. Here, a requirement of 75 percent of samples is specified:

```
$runningTasksSample = $RunningTasks.GetSample(60 * TimeInterval_Second, 120 * TimeInterval_Second, 75);
```

It is also important, due to the previously mentioned delay in sample availability, to always specify a time range with a look-back start time that is older than one minute. This is because it takes approximately one minute for samples to propagate through the system, so samples in the range

`(0 * TimeInterval_Second, 60 * TimeInterval_Second)` will often not be available. Again, you can use the percentage parameter of `GetSample()` to force a particular sample percentage requirement.

IMPORTANT

We **strongly recommend** that you **avoid relying only on** `GetSample(1)` **in your autoscale formulas**. This is because `GetSample(1)` essentially says to the Batch service, "Give me the last sample you have, no matter how long ago you got it." Since it is only a single sample, and it may be an older sample, it may not be representative of the larger picture of recent task or resource state. If you do use `GetSample(1)`, make sure that it's part of a larger statement and not the only data point that your formula relies on.

Metrics

You can use both **resource** and **task** metrics when you're defining a formula. You adjust the target number of dedicated nodes in the pool based on the metrics data that you obtain and evaluate. See the [Variables](#) section above for more information on each metric.

METRIC	DESCRIPTION
--------	-------------

<p>Resource</p>	<p>Resource metrics are based on the CPU, bandwidth, and memory usage of compute nodes, as well as the number of nodes.</p> <p>These service-defined variables are useful for making adjustments based on node count:</p> <ul style="list-style-type: none"> • \$TargetDedicated • \$CurrentDedicated • \$SampleNodeCount <p>These service-defined variables are useful for making adjustments based on node resource usage:</p> <ul style="list-style-type: none"> • \$CPUPercent • \$WallClockSeconds • \$MemoryBytes • \$DiskBytes • \$DiskReadBytes • \$DiskWriteBytes • \$DiskReadOps • \$DiskWriteOps • \$NetworkInBytes • \$NetworkOutBytes
<p>Task</p>	<p>Task metrics are based on the status of tasks, such as Active, Pending, and Completed. The following service-defined variables are useful for making pool-size adjustments based on task metrics:</p> <ul style="list-style-type: none"> • \$ActiveTasks • \$RunningTasks • \$PendingTasks • \$SucceededTasks • \$FailedTasks

Write an autoscale formula

You build an autoscale formula by forming statements that use the above components, then combine those statements into a complete formula. In this section, we'll create an example autoscale formula that can perform some real-world scaling decisions.

First, let's define the requirements for our new autoscale formula. The formula should:

1. **Increase** the target number of compute nodes in a pool if CPU usage is high.
2. **Decrease** the target number of compute nodes in a pool when CPU usage is low.
3. Always restrict the **maximum** number of nodes to 400.

To *increase* the number of nodes during high CPU usage, we define the statement that populates a user-defined variable (`$totalNodes`) with a value that is 110 percent of the current target number of nodes, but only if the minimum average CPU usage during the last 10 minutes was above 70 percent. Otherwise, we use the current dedicated value.

```
$totalNodes =
  (min($CPUPercent.GetSample(TimeInterval_Minute * 10)) > 0.7) ?
  ($CurrentDedicated * 1.1) : $CurrentDedicated;
```

To *decrease* the number of nodes during low CPU usage, the next statement in our formula sets the same

`$totalNodes` variable to 90 percent of the current target number of nodes if the average CPU usage in the past 60 minutes was under 20 percent. Otherwise, use the current value of `$totalNodes` that we populated in the statement above.

```
$totalNodes =  
    (avg($CPUPercent.GetSample(TimeInterval_Minute * 60)) < 0.2) ?  
    ($CurrentDedicated * 0.9) : $totalNodes;
```

Now limit the target number of dedicated compute nodes to a **maximum** of 400:

```
$TargetDedicated = min(400, $totalNodes)
```

Here's the complete formula:

```
$totalNodes =  
    (min($CPUPercent.GetSample(TimeInterval_Minute * 10)) > 0.7) ?  
    ($CurrentDedicated * 1.1) : $CurrentDedicated;  
$totalNodes =  
    (avg($CPUPercent.GetSample(TimeInterval_Minute * 60)) < 0.2) ?  
    ($CurrentDedicated * 0.9) : $totalNodes;  
$TargetDedicated = min(400, $totalNodes)
```

Create an autoscale-enabled pool

To create a new pool with autoscaling enabled, you can use one of the following techniques:

Batch .NET

1. Create the pool with [BatchClient.PoolOperations.CreatePool](#).
2. Set the [CloudPool.AutoScaleEnabled](#) property to `true`.
3. Set the [CloudPool.AutoScaleFormula](#) property with your autoscale formula.
4. (Optional) Set the [CloudPool.AutoScaleEvaluationInterval](#) property (default is 15 minutes).
5. Commit the pool with [CloudPool.Commit](#) or [CommitAsync](#).

Batch REST API

- [Add a pool to an account](#): Specify the `enableAutoScale` and `autoScaleFormula` elements in your REST API request to configure automatic scaling for a pool when you create it.

The following code snippet creates an autoscale-enabled pool by using the [Batch .NET](#) library. The pool's autoscale formula sets the target number of nodes to 5 on Mondays, and 1 on every other day of the week. The [automatic scaling interval](#) is set to 30 minutes. In this and the other C# snippets in this article, "myBatchClient" is a properly initialized instance of [BatchClient](#).

```
CloudPool pool = myBatchClient.PoolOperations.CreatePool("mypool", "3", "small");  
pool.AutoScaleEnabled = true;  
pool.AutoScaleFormula = "$TargetDedicated = (time().weekday == 1 ? 5:1);";  
pool.AutoScaleEvaluationInterval = TimeSpan.FromMinutes(30);  
pool.Commit();
```

In addition to the Batch REST API and .NET SDK, you can use any of the other [Batch SDKs](#), [Batch PowerShell cmdlets](#), and the [Batch CLI](#) to work with autoscaling.

IMPORTANT

When you create an autoscale-enabled pool, you must **not** specify the `targetDedicated` parameter. Also, if you want to manually resize an autoscale-enabled pool (for example, with `BatchClient.PoolOperations.ResizePool`), then you must first **disable** automatic scaling on the pool, then resize it.

Automatic scaling interval

By default, the Batch service adjusts a pool's size according to its autoscale formula every **15 minutes**. This interval is configurable, however, by using the following pool properties:

- `CloudPool.AutoScaleEvaluationInterval` (Batch .NET)
- `autoScaleEvaluationInterval` (REST API)

The minimum interval is five minutes, and the maximum is 168 hours. If an interval outside this range is specified, the Batch service will return a Bad Request (400) error.

NOTE

Autoscaling is not currently intended to respond to changes in less than a minute, but rather is intended to adjust the size of your pool gradually as you run a workload.

Enable autoscaling on an existing pool

If you've already created a pool with a set number of compute nodes by using the `targetDedicated` parameter, you can still enable autoscaling on the pool. Each Batch SDK provides an "enable autoscale" operation, for example:

- `BatchClient.PoolOperations.EnableAutoScale` (Batch .NET)
- `Enable automatic scaling on a pool` (REST API)

When you enable autoscaling on an existing pool, the following applies:

- If automatic scaling is currently **disabled** on the pool when you issue the "enable autoscale" request, you *must* specify a valid autoscale formula when you issue the request. You can *optionally* specify an autoscale evaluation interval. If you do not specify an interval, the default value of 15 minutes is used.
- If autoscale is currently **enabled** on the pool, you can specify an autoscale formula, an evaluation interval, or both. You can't omit both properties.
 - If you specify a new autoscale evaluation interval, then the existing evaluation schedule is stopped and a new schedule is started. The new schedule's start time is the time at which the "enable autoscale" request was issued.
 - If you omit either the autoscale formula or evaluation interval, the Batch service continues to use the current value of that setting.

NOTE

If a value was specified for the `targetDedicated` parameter when the pool was created, it is ignored when the automatic scaling formula is evaluated.

This C# code snippet uses the `Batch .NET` library to enable autoscaling on an existing pool:

```
// Define the autoscaling formula. This formula sets the target number of nodes
// to 5 on Mondays, and 1 on every other day of the week
string myAutoScaleFormula = "$TargetDedicated = (time().weekday == 1 ? 5:1)";

// Set the autoscale formula on the existing pool
myBatchClient.PoolOperations.EnableAutoScale(
    "myexistingpool",
    autoscaleFormula: myAutoScaleFormula);
```

Update an autoscale formula

You use the same "enable autoscale" request to *update* the formula on an existing autoscale-enabled pool (for example, with [EnableAutoScale](#) in Batch .NET). There is no special "update autoscale" operation. For example, if autoscaling is already enabled on "myexistingpool" when the following code is executed, its autoscale formula is replaced with the contents of `myNewFormula`.

```
myBatchClient.PoolOperations.EnableAutoScale(
    "myexistingpool",
    autoscaleFormula: myNewFormula);
```

Update the autoscale interval

As with updating an autoscale formula, you use the same [EnableAutoScale](#) method to change the autoscale evaluation interval of an existing autoscale-enabled pool. For example, to set the autoscale evaluation interval to 60 minutes for a pool that's already autoscale-enabled:

```
myBatchClient.PoolOperations.EnableAutoScale(
    "myexistingpool",
    autoscaleEvaluationInterval: TimeSpan.FromMinutes(60));
```

Evaluate an autoscale formula

You can evaluate a formula before applying it to a pool. In this way, you can perform a "test run" of the formula to see how its statements evaluate before you put the formula into production.

To evaluate an autoscale formula, you must first **enable autoscaling** on the pool with a **valid formula**. If you want to test a formula on a pool that doesn't yet have autoscaling enabled, you can use the one-line formula `$TargetDedicated = 0` when you first enable autoscaling. Then, use one of the following to evaluate the formula you want to test:

- [BatchClient.PoolOperations.EvaluateAutoScale](#) or [EvaluateAutoScaleAsync](#)

These Batch .NET methods require the ID of an existing pool and a string containing the autoscale formula to evaluate. The evaluation results are contained in the returned [AutoScaleEvaluation](#) instance.

- [Evaluate an automatic scaling formula](#)

In this REST API request, specify the pool ID in the URI, and the autoscale formula in the *autoScaleFormula* element of the request body. The response of the operation contains any error information that might be related to the formula.

In this [Batch .NET](#) code snippet, we evaluate a formula prior to applying it to the [CloudPool](#). If the pool does not have autoscaling enabled, we enable it first.

```

// First obtain a reference to an existing pool
CloudPool pool = batchClient.PoolOperations.GetPool("myExistingPool");

// If autoscaling isn't already enabled on the pool, enable it.
// You can't evaluate an autoscale formula on non-autoscale-enabled pool.
if (pool.AutoScaleEnabled == false)
{
    // We need a valid autoscale formula to enable autoscaling on the
    // pool. This formula is valid, but won't resize the pool:
    pool.EnableAutoScale(
        autoscaleFormula: $"$TargetDedicated = {pool.CurrentDedicated};",
        autoscaleEvaluationInterval: TimeSpan.FromMinutes(5));

    // Batch limits EnableAutoScale calls to once every 30 seconds.
    // Because we want to apply our new autoscale formula below if it
    // evaluates successfully, and we *just* enabled autoscaling on
    // this pool, we pause here to ensure we pass that threshold.
    Thread.Sleep(TimeSpan.FromSeconds(31));

    // Refresh the properties of the pool so that we've got the
    // latest value for AutoScaleEnabled
    pool.Refresh();
}

// We must ensure that autoscaling is enabled on the pool prior to
// evaluating a formula
if (pool.AutoScaleEnabled == true)
{
    // The formula to evaluate - adjusts target number of nodes based on
    // day of week and time of day
    string myFormula = @"
        $curTime = time();
        $workHours = $curTime.hour >= 8 && $curTime.hour < 18;
        $isWeekday = $curTime.weekday >= 1 && $curTime.weekday <= 5;
        $isWorkingWeekdayHour = $workHours && $isWeekday;
        $TargetDedicated = $isWorkingWeekdayHour ? 20:10;
    ";

    // Perform the autoscale formula evaluation. Note that this does not
    // actually apply the formula to the pool.
    AutoScaleRun eval =
        batchClient.PoolOperations.EvaluateAutoScale(pool.Id, myFormula);

    if (eval.Error == null)
    {
        // Evaluation success - print the results of the AutoScaleRun.
        // This will display the values of each variable as evaluated by the
        // autoscale formula.
        Console.WriteLine("AutoScaleRun.Results: " +
            eval.Results.Replace("$", "\n    $"));

        // Apply the formula to the pool since it evaluated successfully
        batchClient.PoolOperations.EnableAutoScale(pool.Id, myFormula);
    }
    else
    {
        // Evaluation failed, output the message associated with the error
        Console.WriteLine("AutoScaleRun.Error.Message: " +
            eval.Error.Message);
    }
}
}

```

Successful evaluation of the formula in this snippet will result in output similar to the following:

```
AutoScaleRun.Results:
    $TargetDedicated=10;
    $NodeDeallocationOption=requeue;
    $curTime=2016-10-13T19:18:47.805Z;
    $isWeekday=1;
    $isWorkingWeekdayHour=0;
    $workHours=0
```

Get information about autoscale runs

To ensure your formula is performing as expected, we recommend you periodically check the results of the autoscaling "runs" Batch performs on your pool. To do so, get (or refresh) a reference to the pool, and examine the properties of its last autoscale run.

In Batch .NET, the [CloudPool.AutoScaleRun](#) property has several properties providing information about the latest automatic scaling run performed on the pool by the Batch service.

- [AutoScaleRun.Timestamp](#)
- [AutoScaleRun.Results](#)
- [AutoScaleRun.Error](#)

In the REST API, the [Get information about a pool](#) request returns information about the pool, which includes the latest automatic scaling run information in [autoScaleRun](#).

The following C# code snippet uses the Batch .NET library to print information about the last autoscaling run on pool "myPool":

```
Cloud pool = myBatchClient.PoolOperations.GetPool("myPool");
Console.WriteLine("Last execution: " + pool.AutoScaleRun.Timestamp);
Console.WriteLine("Result:" + pool.AutoScaleRun.Results.Replace("$", "\\n $"));
Console.WriteLine("Error: " + pool.AutoScaleRun.Error);
```

Sample output of the preceding snippet:

```
Last execution: 10/14/2016 18:36:43
Result:
    $TargetDedicated=10;
    $NodeDeallocationOption=requeue;
    $curTime=2016-10-14T18:36:43.282Z;
    $isWeekday=1;
    $isWorkingWeekdayHour=0;
    $workHours=0
Error:
```

Example autoscale formulas

Let's take a look at a few formulas that show different ways to adjust the amount of compute resources in a pool.

Example 1: Time-based adjustment

Perhaps you want to adjust the pool size based on the day of the week and time of day, to increase or decrease the number of nodes in the pool accordingly.

This formula first obtains the current time. If it's a weekday (1-5) and within working hours (8 AM to 6 PM), the target pool size is set to 20 nodes. Otherwise, it's set to 10 nodes.

```

$curTime = time();
$workHours = $curTime.hour >= 8 && $curTime.hour < 18;
$isWeekday = $curTime.weekday >= 1 && $curTime.weekday <= 5;
$isWorkingWeekdayHour = $workHours && $isWeekday;
$TargetDedicated = $isWorkingWeekdayHour ? 20:10;

```

Example 2: Task-based adjustment

In this example, the pool size is adjusted based on the number of tasks in the queue. Note that both comments and line breaks are acceptable in formula strings.

```

// Get pending tasks for the past 15 minutes.
$samples = $ActiveTasks.GetSamplePercent(TimeInterval_Minute * 15);
// If we have fewer than 70 percent data points, we use the last sample point,
// otherwise we use the maximum of last sample point and the history average.
$tasks = $samples < 70 ? max(0,$ActiveTasks.GetSample(1)) : max( $ActiveTasks.GetSample(1),
avg($ActiveTasks.GetSample(TimeInterval_Minute * 15)));
// If number of pending tasks is not 0, set targetVM to pending tasks, otherwise
// half of current dedicated.
$targetVMs = $tasks > 0? $tasks:max(0, $TargetDedicated/2);
// The pool size is capped at 20, if target VM value is more than that, set it
// to 20. This value should be adjusted according to your use case.
$TargetDedicated = max(0, min($targetVMs, 20));
// Set node deallocation mode - keep nodes active only until tasks finish
$NodeDeallocationOption = taskcompletion;

```

Example 3: Accounting for parallel tasks

This is another example that adjusts the pool size based on the number of tasks. This formula also takes into account the [MaxTasksPerComputeNode](#) value that has been set for the pool. This is particularly useful in situations where [parallel task execution](#) has been enabled on your pool.

```

// Determine whether 70 percent of the samples have been recorded in the past
// 15 minutes; if not, use last sample
$samples = $ActiveTasks.GetSamplePercent(TimeInterval_Minute * 15);
$tasks = $samples < 70 ? max(0,$ActiveTasks.GetSample(1)) : max(
$ActiveTasks.GetSample(1),avg($ActiveTasks.GetSample(TimeInterval_Minute * 15)));
// Set the number of nodes to add to one-fourth the number of active tasks (the
// MaxTasksPerComputeNode property on this pool is set to 4, adjust this number
// for your use case)
$cores = $TargetDedicated * 4;
$extraVMs = (($tasks - $cores) + 3) / 4;
$targetVMs = ($TargetDedicated + $extraVMs);
// Attempt to grow the number of compute nodes to match the number of active
// tasks, with a maximum of 3
$TargetDedicated = max(0,min($targetVMs,3));
// Keep the nodes active until the tasks finish
$NodeDeallocationOption = taskcompletion;

```

Example 4: Setting an initial pool size

This example shows a C# code snippet with an autoscale formula that sets the pool size to a certain number of nodes for an initial time period. Then it adjusts the pool size based on the number of running and active tasks after the initial time period has elapsed.

The formula in the following code snippet:

- Sets the initial pool size to four nodes.
- Does not adjust the pool size within the first 10 minutes of the pool's lifecycle.
- After 10 minutes, obtains the max value of the number of running and active tasks within the past 60 minutes.
 - If both values are 0 (indicating that no tasks were running or active in the last 60 minutes), the pool size

is set to 0.

- If either value is greater than zero, no change is made.

```
string now = DateTime.UtcNow.ToString("r");
string formula = string.Format(@"
    $TargetDedicated = {1};
    lifespan          = time() - time("{0}");
    span              = TimeInterval_Minute * 60;
    startup            = TimeInterval_Minute * 10;
    ratio              = 50;

    $TargetDedicated = (lifespan > startup ? (max($RunningTasks.GetSample(span, ratio),
$ActiveTasks.GetSample(span, ratio)) == 0 ? 0 : $TargetDedicated) : {1});
    ", now, 4);
```

Next steps

- [Maximize Azure Batch compute resource usage with concurrent node tasks](#) contains details about how you can execute multiple tasks simultaneously on the compute nodes in your pool. In addition to autoscaling, this feature may help to lower job duration for some workloads, saving you money.
- For another efficiency booster, ensure that your Batch application queries the Batch service in the most optimal way. In [Query the Azure Batch service efficiently](#), you'll learn how to limit the amount of data that crosses the wire when you query the status of potentially thousands of compute nodes or tasks.

Authenticate from Batch solutions with Active Directory

3/17/2017 • 10 min to read • [Edit Online](#)

Azure Batch supports authentication with [Azure Active Directory](#) (Azure AD) for the Batch service and the Batch management service. Azure AD is Microsoft's multi-tenant cloud based directory and identity management service. Azure itself uses Azure AD for the authentication of its customers, service administrators, and organizational users.

In this article, we explore using Azure AD to authenticate from applications that use the Batch Management .NET library or the Batch .NET library. In the context of the Batch .NET APIs, we show how to use Azure AD to authenticate a subscription administrator or co-administrator, using integrated authentication. The authenticated user can then issue requests to Azure Batch.

It's also possible to use Azure AD to authenticate access to an application running unattended. Here we focus on using Azure AD integrated authentication, and refer you to other resources to learn about authenticating unattended applications.

Use Azure AD with Batch management solutions

The Batch Management .NET library exposes types for working with Batch accounts, account keys, applications, and application packages. The Batch Management .NET library is an Azure resource provider client, and is used together with [Azure Resource Manager](#) to manage these resources programmatically.

Azure AD is required to authenticate requests made through any Azure resource provider client, including the Batch Management .NET library, and through [Azure Resource Manager](#).

In this section, we use the [AccountManagement](#) sample project, available on GitHub, to walk through using Azure AD with the Batch Management .NET library. The AccountManagement sample is a console application that accesses a subscription programmatically, creates a resource group and a new Batch account, and performs some operations on the account.

To learn more about using the Batch Management .NET library and the AccountManagement sample, see [Manage Batch accounts and quotas with the Batch Management client library for .NET](#).

Register your application with Azure AD

The Azure [Active Directory Authentication Library](#) (ADAL) provides a programmatic interface to Azure AD for use within your applications. To call ADAL from your application, you must register your application in an Azure AD tenant. When you register your application, you supply Azure AD with information about your application, including a name for it within the Azure AD tenant. Azure AD then provides an application ID that you use to associate your application with Azure AD at runtime. To learn more about the application ID, see [Application and service principal objects in Azure Active Directory](#).

To register the AccountManagement sample application, follow the steps in the [Adding an Application](#) section in [Integrating applications with Azure Active Directory](#). Specify **Native Client Application** for the type of application. For the **Redirect URI**, you can specify any valid URI (such as `http://myaccountmanagementsample`), as it does not need to be a real endpoint:

Create

PREVIEW

*

Name

BatchManagementSample

✓

Application Type

Native

*

Redirect URI

http://myaccountmanagementsample

✓

Once you complete the registration process, you'll see the application ID and the object (service principal) ID listed for your application.

BatchManagementSample

Registered app - PREVIEW

Settings

Manifest

Delete

Essentials

Display Name

BatchManagementSample

Application Type

Native

Home Page

Application ID

c1ec2f68-b330-48ea-99b1-f0d24a8debe8

Object ID

40abb301-77e0-40d1-9f70-00699a36508a

Managed Application In Local Directory

BatchManagementSample

All settings →

Update your code to reference your application ID

Your client application uses the application ID (also referred to as the client ID) to access Azure AD at runtime. Once you've registered your application in the Azure portal, update your code to use the application ID provided by Azure AD for your registered application. In the AccountManagement sample application, copy your application ID from the Azure portal to the appropriate constant:

```
// Specify the unique identifier (the "Client ID") for your application. This is required so that your
// native client application (i.e. this sample) can access the Microsoft Azure AD Graph API. For information
// about registering an application in Azure Active Directory, please see "Adding an Application" here:
// https://azure.microsoft.com/documentation/articles/active-directory-integrating-applications/
private const string ClientId = "<application-id>;
```

Also copy the redirect URI that you specified during the registration process.

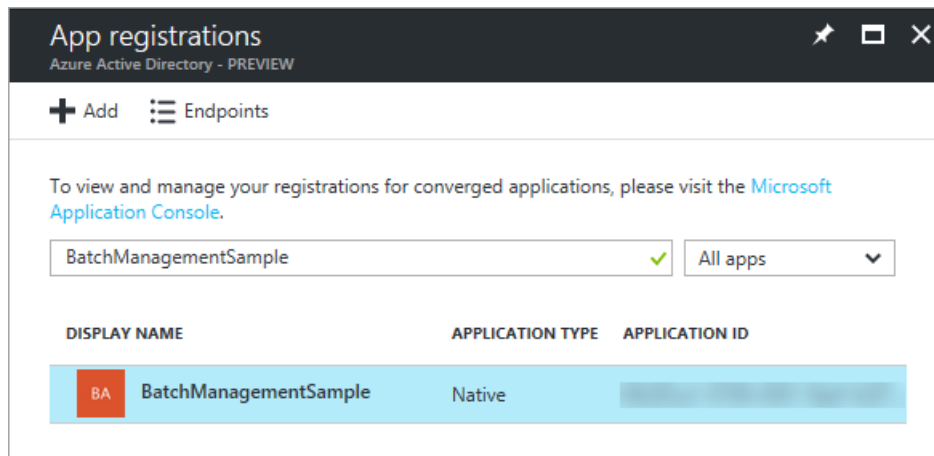
```
// The URI to which Azure AD will redirect in response to an OAuth 2.0 request. This value is
// specified by you when you register an application with AAD (see ClientId comment). It does not
// need to be a real endpoint, but must be a valid URI (e.g. https://accountmgmtsapp).
private const string RedirectUri = "http://myaccountmanagementsample";
```

Grant the Azure Resource Manager API access to your application

Next, you'll need to delegate access to your application to the Azure Resource Manager API. The Azure AD identifier for the Resource Manager API is **Windows Azure Service Management API**.

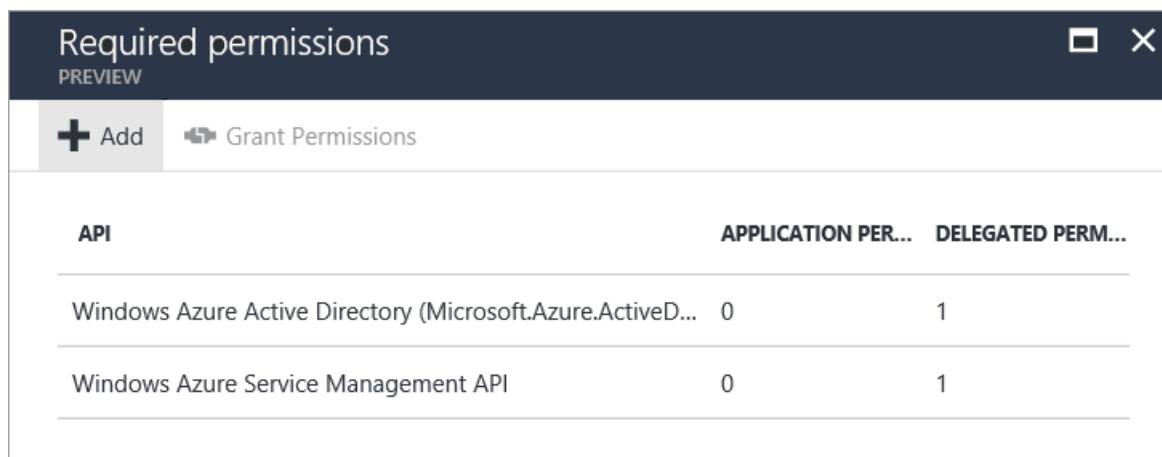
Follow these steps in the Azure portal:

1. In the left-hand navigation pane of the Azure portal, choose **More Services**, click **App Registrations**, and click **Add**.
2. Search for the name of your application in the list of app registrations:



3. Display the **Settings** blade. In the **API Access** section, select **Required permissions**.
4. Click **Add** to add a new required permission.
5. In step 1, enter **Windows Azure Service Management API**, select that API from the list of results, and click the **Select** button.
6. In step 2, select the check box next to **Access Azure classic deployment model as organization users**, and click the **Select** button.
7. Click the **Done** button.

The **Required Permissions** blade now shows that permissions to your application are granted to both the ADAL and Resource Manager APIs. Permissions are granted to ADAL by default when you first register your app with Azure AD.



Acquire an Azure AD authentication token

The AccountManagement sample application defines constants that provide the endpoint for Azure AD and for Azure Resource Manager. The sample application uses these constants to query Azure AD for subscription information. Leave these constants unchanged:

```
// Azure Active Directory "common" endpoint.  
private const string AuthorityUri = "https://login.microsoftonline.com/common";  
// Azure service management resource  
private const string ResourceUri = "https://management.core.windows.net/";
```

After you register the AccountManagement sample in the Azure AD tenant and provide the necessary values within the sample source code, the sample is ready to authenticate using Azure AD. When you run the sample, the ADAL attempts to acquire an authentication token. At this step, it prompts you for your Microsoft credentials:

```
// Obtain an access token using the "common" AAD resource. This allows the application
// to query AAD for information that lies outside the application's tenant (such as for
// querying subscription information in your Azure account).
AuthenticationContext authContext = new AuthenticationContext(AuthorityUri);
AuthenticationResult authResult = authContext.AcquireToken(ResourceUri,
    ClientId,
    new Uri(RedirectUri),
    PromptBehavior.Auto);
```

After you provide your credentials, the sample application can proceed to issue authenticated requests to the Batch management service.

Use Azure AD with Batch service solutions

The Batch .NET library provides types for building parallel processing workflows with the Batch service. The Batch service supports both [Shared Key](#) authentication and authentication through Azure AD. In this section, we discuss authentication via Azure AD.

NOTE

When you create a Batch account, you can specify whether pools are to be allocated in a subscription managed by Batch, or in a user subscription. If your account allocates pools in a user subscription, then you must use Azure AD to authenticate requests to resources in that account.

Authenticating Batch .NET applications via Azure AD is similar to authenticating Batch Management .NET applications. There are a few differences, described in this section.

Batch service endpoints

The Batch service endpoints differ from those that you use with Batch Management .NET.

The Azure AD endpoint for the Batch service is:

```
https://login.microsoftonline.com/common
```

The resource endpoint for the Batch service is:

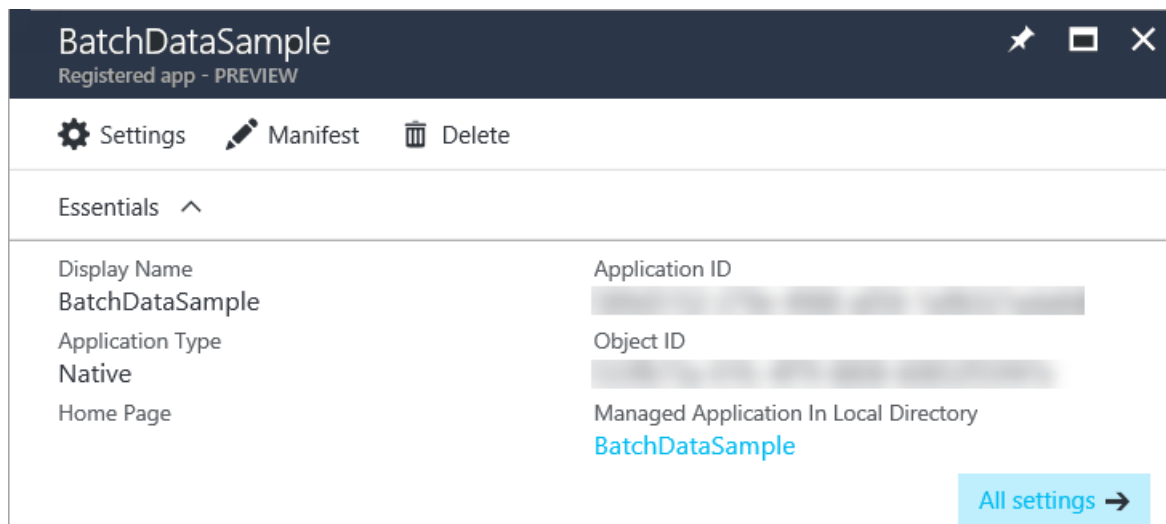
```
https://batch.core.windows.net/
```

Grant the Batch service API access to your application

Before you can authenticate via Azure AD from your Batch application, you need to register your application with Azure AD and grant access to the Batch service API. The Azure AD identifier for the Batch service API is **Microsoft Azure Batch (MicrosoftAzureBatch)**.

1. To register your Batch application, follow the steps in the [Adding an Application](#) section in [Integrating applications with Azure Active Directory](#). For the **Redirect URI**, you can specify any valid URI. It does not need to be a real endpoint.

After you've registered your application, you'll see the application ID and object ID:



- Next, display the **Settings** blade. In the **API Access** section, select **Required permissions**.
- In the **Required permissions** blade, click the **Add** button.
- In step 1, search for **MicrosoftAzureBatch**, select **Microsoft Azure Batch (MicrosoftAzureBatch)**, and click the **Select** button.
- In step 2, select the check box next to **Access Azure Batch Service** and click the **Select** button.
- Click the **Done** button.

The **Required Permissions** blade now shows that your Azure AD application grants access to both the Azure AD and Azure Batch APIs.

Required permissions		
PREVIEW		
+ Add Grant Permissions		
API	APPLICATION PER...	DELEGATED PERM...
Windows Azure Active Directory (Microsoft.Azure.ActiveD...	0	1
Microsoft Azure Batch (MicrosoftAzureBatch)	0	1

Authentication for Batch accounts in a user subscription

When you create a new Batch account, you can choose the subscription in which pools are allocated. Your choice affects how you authenticate requests made to resources in that account

By default, Batch pools are allocated in a Batch service subscription. If you choose this option, you can authenticate requests to resources in that account with either Shared Key or with Azure AD.

You can also specify that Batch pools are allocated in a specified user subscription. If you choose this option, you must authenticate with Azure AD.

Best practices for using Azure AD with Batch

An Azure AD authentication token expires after one hour. When using a long-lived **BatchClient** object, we recommend that you retrieve a token from ADAL on every request to ensure you always have a valid token.

To achieve this in .NET, write a method that retrieves the token from Azure AD and pass that method to a **BatchTokenCredentials** object as a delegate. The delegate method is called on every request to the Batch service to ensure that a valid token is provided. By default ADAL caches tokens, so a new token is retrieved from Azure AD

only when necessary. For an example, see [Code example: Using Azure AD with Batch .NET](#) in the next section. For more information about tokens in Azure AD, see [Authentication Scenarios for Azure AD](#).

Code example: Using Azure AD with Batch .NET

To write Batch .NET code that authenticates with Azure AD, reference the [Azure Batch .NET](#) package and the [ADAL](#) package.

Include the following `using` statements in your code:

```
using Microsoft.Azure.Batch;  
using Microsoft.Azure.Batch.Auth;  
using Microsoft.IdentityModel.Clients.ActiveDirectory;
```

Reference the Azure AD common endpoint and the Azure AD endpoint for the Batch service in your code:

```
private const string AuthorityUri = "https://login.microsoftonline.com/common";  
private const string BatchResourceUri = "https://batch.core.windows.net/";
```

Reference your Batch account endpoint:

```
private const string BatchAccountEndpoint = "https://myaccount.westcentralus.batch.azure.com";
```

Specify the application ID (client ID) for your application. The application ID is available from your app registration in the Azure portal; see the section titled [Grant the Batch service API access to your application](#) to retrieve it.

```
private const string ClientId = "<application-id>";
```

Also specify a redirect URI, which can be any valid URI.

```
private const string RedirectUri = "http://mybatchdatasample";
```

Write a callback method to acquire the authentication token from Azure AD. The **AcquireTokenAsync** method prompts the user for their credentials and uses those credentials to acquire a new token.

```
public static async Task<string> GetAuthenticationTokenAsync()  
{  
    var authContext = new AuthenticationContext(AuthorityUri);  
  
    // Acquire the authentication token from Azure AD.  
    var authResult = await authContext.AcquireTokenAsync(BatchResourceUri,  
                                                         ClientId,  
                                                         new Uri(RedirectUri),  
                                                         new PlatformParameters(PromptBehavior.Auto));  
  
    return authResult.AccessToken;  
}
```

Construct a **BatchTokenCredentials** object that takes the delegate as a parameter. Use those credentials to open a **BatchClient** object. You can then use that **BatchClient** object for subsequent operations against the Batch service.

```
public static async Task PerformBatchOperations()
{
    Func<Task<string>> tokenProvider = () => GetAuthenticationTokenAsync();

    using (var client = await BatchClient.OpenAsync(new BatchTokenCredentials(BatchAccountEndpoint,
    tokenProvider)))
    {
        await client.JobOperations.ListJobs().ToListAsync();
    }
}
```

The **GetAuthenticationTokenAsync** callback method shown above uses Azure AD for integrated authentication of a user who is interacting with the application. The call to the **AcquireTokenAsync** method prompts the user for their credentials, and the application proceeds once the user provides them. You can also use Azure AD to authenticate an unattended application by using an Azure AD service principal. For more information, see [Application and service principal objects in Azure Active Directory](#) and [Use portal to create Active Directory application and service principal that can access resources](#).

Next steps

For more information on running the [AccountManagement sample application](#), see [Manage Batch accounts and quotas with the Batch Management client library for .NET](#).

To learn more about Azure AD, see the [Azure Active Directory Documentation](#). In-depth examples showing how to use ADAL are available in the [Azure Code Samples](#) library.

Run tasks concurrently to maximize usage of Batch compute nodes

2/27/2017 • 5 min to read • [Edit Online](#)

By running more than one task simultaneously on each compute node in your Azure Batch pool, you can maximize resource usage on a smaller number of nodes in the pool. For some workloads, this can result in shorter job times and lower cost.

While some scenarios benefit from dedicating all of a node's resources to a single task, several situations benefit from allowing multiple tasks to share those resources:

- **Minimizing data transfer** when tasks are able to share data. In this scenario, you can dramatically reduce data transfer charges by copying shared data to a smaller number of nodes and executing tasks in parallel on each node. This especially applies if the data to be copied to each node must be transferred between geographic regions.
- **Maximizing memory usage** when tasks require a large amount of memory, but only during short periods of time, and at variable times during execution. You can employ fewer, but larger, compute nodes with more memory to efficiently handle such spikes. These nodes would have multiple tasks running in parallel on each node, but each task would take advantage of the nodes' plentiful memory at different times.
- **Mitigating node number limits** when inter-node communication is required within a pool. Currently, pools configured for inter-node communication are limited to 50 compute nodes. If each node in such a pool is able to execute tasks in parallel, a greater number of tasks can be executed simultaneously.
- **Replicating an on-premises compute cluster**, such as when you first move a compute environment to Azure. If your current on-premises solution executes multiple tasks per compute node, you can increase the maximum number of node tasks to more closely mirror that configuration.

Example scenario

As an example to illustrate the benefits of parallel task execution, let's say that your task application has CPU and memory requirements such that [Standard_D1](#) nodes are sufficient. But, in order to finish the job in the required time, 1,000 of these nodes are needed.

Instead of using [Standard_D1](#) nodes that have 1 CPU core, you could use [Standard_D14](#) nodes that have 16 cores each, and enable parallel task execution. Therefore, *16 times fewer nodes* could be used--instead of 1,000 nodes, only 63 would be required. Additionally, if large application files or reference data are required for each node, job duration and efficiency are again improved since the data is copied to only 16 nodes.

Enable parallel task execution

You configure compute nodes for parallel task execution at the pool level. With the Batch .NET library, set the [CloudPool.MaxTasksPerComputeNode](#) property when you create a pool. If you are using the Batch REST API, set the [maxTasksPerNode](#) element in the request body during pool creation.

Azure Batch allows you to set maximum tasks per node up to four times (4x) the number of node cores. For example, if the pool is configured with nodes of size "Large" (four cores), then `maxTasksPerNode` may be set to 16. For details on the number of cores for each of the node sizes, see [Sizes for Cloud Services](#). For more information on service limits, see [Quotas and limits for the Azure Batch service](#).

TIP

Be sure to take into account the `maxTasksPerNode` value when you construct an [autoscale formula](#) for your pool. For example, a formula that evaluates `$RunningTasks` could be dramatically affected by an increase in tasks per node. See [Automatically scale compute nodes in an Azure Batch pool](#) for more information.

Distribution of tasks

When the compute nodes in a pool can execute tasks concurrently, it's important to specify how you want the tasks to be distributed across the nodes in the pool.

By using the [CloudPool.TaskSchedulingPolicy](#) property, you can specify that tasks should be assigned evenly across all nodes in the pool ("spreading"). Or you can specify that as many tasks as possible should be assigned to each node before tasks are assigned to another node in the pool ("packing").

As an example of how this feature is valuable, consider the pool of [Standard_D14](#) nodes (in the example above) that is configured with a [CloudPool.MaxTasksPerComputeNode](#) value of 16. If the [CloudPool.TaskSchedulingPolicy](#) is configured with a [ComputeNodeFillType](#) of *Pack*, it would maximize usage of all 16 cores of each node and allow an [autoscaling pool](#) to prune unused nodes from the pool (nodes without any tasks assigned). This minimizes resource usage and saves money.

Batch .NET example

This [Batch .NET](#) API code snippet shows a request to create a pool that contains four large nodes with a maximum of four tasks per node. It specifies a task scheduling policy that will fill each node with tasks prior to assigning tasks to another node in the pool. For more information on adding pools by using the Batch .NET API, see [BatchClient.PoolOperations.CreatePool](#).

```
CloudPool pool =
    batchClient.PoolOperations.CreatePool(
        poolId: "mypool",
        targetDedicated: 4
        virtualMachineSize: "large",
        cloudServiceConfiguration: new CloudServiceConfiguration(osFamily: "4"));

pool.MaxTasksPerComputeNode = 4;
pool.TaskSchedulingPolicy = new TaskSchedulingPolicy(ComputeNodeFillType.Pack);
pool.Commit();
```

Batch REST example

This [Batch REST](#) API snippet shows a request to create a pool that contains two large nodes with a maximum of four tasks per node. For more information on adding pools by using the REST API, see [Add a pool to an account](#).


```
{
  "odata.metadata":"https://myaccount.myregion.batch.azure.com/$metadata#pools/@Element",
  "id":"mypool",
  "vmSize":"large",
  "cloudServiceConfiguration": {
    "osFamily":"4",
    "targetOSVersion":"*",
  }
  "targetDedicated":2,
  "maxTasksPerNode":4,
  "enableInterNodeCommunication":true,
}
```

NOTE

You can set the `maxTasksPerNode` element and [MaxTasksPerComputeNode](#) property only at pool creation time. They cannot be modified after a pool has already been created.

Code sample

The [ParallelNodeTasks](#) project on GitHub illustrates the use of the [CloudPool.MaxTasksPerComputeNode](#) property.

This C# console application uses the [Batch .NET](#) library to create a pool with one or more compute nodes. It executes a configurable number of tasks on those nodes to simulate variable load. Output from the application specifies which nodes executed each task. The application also provides a summary of the job parameters and duration. The summary portion of the output from two different runs of the sample application appears below.

```
Nodes: 1
Node size: large
Max tasks per node: 1
Tasks: 32
Duration: 00:30:01.4638023
```

The first execution of the sample application shows that with a single node in the pool and the default setting of one task per node, the job duration is over 30 minutes.

```
Nodes: 1
Node size: large
Max tasks per node: 4
Tasks: 32
Duration: 00:08:48.2423500
```

The second run of the sample shows a significant decrease in job duration. This is because the pool was configured with four tasks per node, which allows for parallel task execution to complete the job in nearly a quarter of the time.

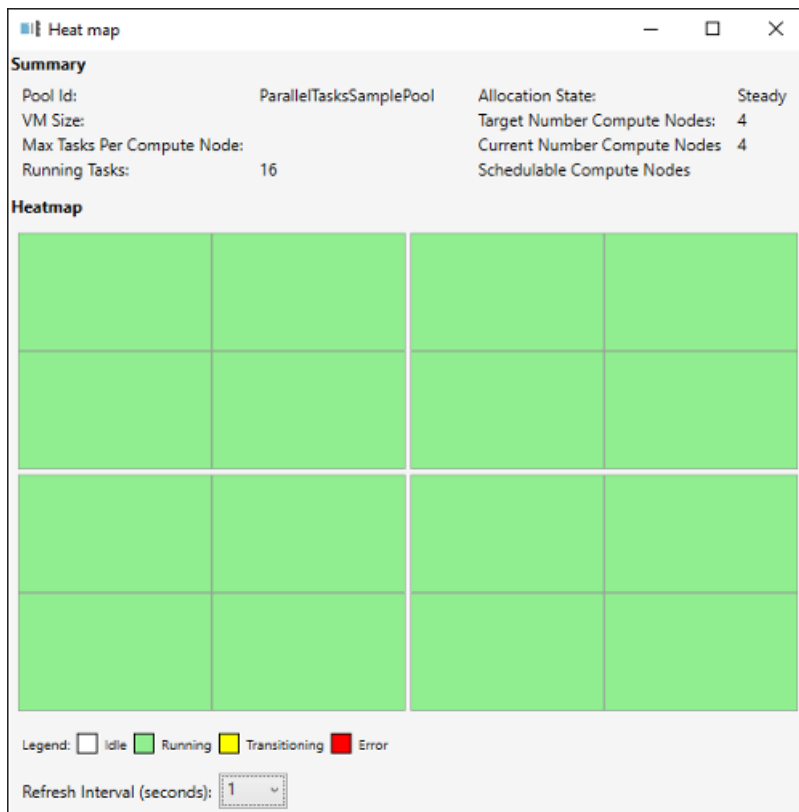
NOTE

The job durations in the summaries above do not include pool creation time. Each of the jobs above was submitted to previously created pools whose compute nodes were in the *Idle* state at submission time.

Next steps

Batch Explorer Heat Map

The [Azure Batch Explorer](#), one of the Azure Batch [sample applications](#), contains a *Heat Map* feature that provides visualization of task execution. When you're executing the [ParallelTasks](#) sample application, you can use the Heat Map feature to easily visualize the execution of parallel tasks on each node.



Batch Explorer Heat Map showing a pool of four nodes, with each node currently executing four tasks

Create queries to list Batch resources efficiently

2/27/2017 • 12 min to read • [Edit Online](#)

Here you'll learn how to increase your Azure Batch application's performance by reducing the amount of data that is returned by the service when you query jobs, tasks, and compute nodes with the [Batch .NET](#) library.

Nearly all Batch applications need to perform some type of monitoring or other operation that queries the Batch service, often at regular intervals. For example, to determine whether there are any queued tasks remaining in a job, you must get data on every task in the job. To determine the status of nodes in your pool, you must get data on every node in the pool. This article explains how to execute such queries in the most efficient way.

Meet the DetailLevel

In a production Batch application, entities like jobs, tasks, and compute nodes can number in the thousands. When you request information on these resources, a potentially large amount of data must "cross the wire" from the Batch service to your application on each query. By limiting the number of items and type of information that is returned by a query, you can increase the speed of your queries, and therefore the performance of your application.

This [Batch .NET](#) API code snippet lists *every* task that is associated with a job, along with *all* of the properties of each task:

```
// Get a collection of all of the tasks and all of their properties for job-001
IPagedEnumerable<CloudTask> allTasks =
    batchClient.JobOperations.ListTasks("job-001");
```

You can perform a much more efficient list query, however, by applying a "detail level" to your query. You do this by supplying an [ODATADetailLevel](#) object to the [JobOperations.ListTasks](#) method. This snippet returns only the ID, command line, and compute node information properties of completed tasks:

```
// Configure an ODATADetailLevel specifying a subset of tasks and
// their properties to return
ODATADetailLevel detailLevel = new ODATADetailLevel();
detailLevel.FilterClause = "state eq 'completed'";
detailLevel.SelectClause = "id,commandLine,nodeInfo";

// Supply the ODATADetailLevel to the ListTasks method
IPagedEnumerable<CloudTask> completedTasks =
    batchClient.JobOperations.ListTasks("job-001", detailLevel);
```

In this example scenario, if there are thousands of tasks in the job, the results from the second query will typically be returned much quicker than the first. More information about using [ODATADetailLevel](#) when you list items with the Batch .NET API is included [below](#).

IMPORTANT

We highly recommend that you *always* supply an [ODATADetailLevel](#) object to your .NET API list calls to ensure maximum efficiency and performance of your application. By specifying a detail level, you can help to lower Batch service response times, improve network utilization, and minimize memory usage by client applications.

Filter, select, and expand

The [Batch .NET](#) and [Batch REST](#) APIs provide the ability to reduce both the number of items that are returned in a list, as well as the amount of information that is returned for each. You do so by specifying **filter**, **select**, and **expand strings** when performing list queries.

Filter

The filter string is an expression that reduces the number of items that are returned. For example, list only the running tasks for a job, or list only compute nodes that are ready to run tasks.

- The filter string consists of one or more expressions, with an expression that consists of a property name, operator, and value. The properties that can be specified are specific to each entity type that you query, as are the operators that are supported for each property.
- Multiple expressions can be combined by using the logical operators `and` and `or`.
- This example filter string lists only the running "render" tasks:

```
(state eq 'running') and startswith(id, 'renderTask')
```

Select

The select string limits the property values that are returned for each item. You specify a list of property names, and only those property values are returned for the items in the query results.

- The select string consists of a comma-separated list of property names. You can specify any of the properties for the entity type you are querying.
- This example select string specifies that only three property values should be returned for each task:

```
id, state, stateTransitionTime
```

Expand

The expand string reduces the number of API calls that are required to obtain certain information. When you use an expand string, more information about each item can be obtained with a single API call. Rather than first obtaining the list of entities, then requesting information for each item in the list, you use an expand string to obtain the same information in a single API call. Less API calls means better performance.

- Similar to the select string, the expand string controls whether certain data is included in list query results.
- The expand string is only supported when it is used in listing jobs, job schedules, tasks, and pools. Currently, it only supports statistics information.
- When all properties are required and no select string is specified, the expand string *must* be used to get statistics information. If a select string is used to obtain a subset of properties, then `stats` can be specified in the select string, and the expand string does not need to be specified.
- This example expand string specifies that statistics information should be returned for each item in the list:

```
stats
```

NOTE

When constructing any of the three query string types (filter, select, and expand), you must ensure that the property names and case match that of their REST API element counterparts. For example, when working with the [.NET CloudTask](#) class, you must specify **state** instead of **State**, even though the .NET property is [CloudTask.State](#). See the tables below for property mappings between the .NET and REST APIs.

Rules for filter, select, and expand strings

- Properties names in filter, select, and expand strings should appear as they do in the [Batch REST](#) API--even when you use [Batch .NET](#) or one of the other Batch SDKs.
- All property names are case-sensitive, but property values are case insensitive.
- Date/time strings can be one of two formats, and must be preceded with `DateTime`.

- W3C-DTF format example: `creationTime gt DateTime'2011-05-08T08:49:37Z'`
- RFC 1123 format example: `creationTime gt DateTime'Sun, 08 May 2011 08:49:37 GMT'`
- Boolean strings are either `true` or `false`.
- If an invalid property or operator is specified, a `400 (Bad Request)` error will result.

Efficient querying in Batch .NET

Within the [Batch .NET](#) API, the [ODATADetailLevel](#) class is used for supplying filter, select, and expand strings to list operations. The [ODATADetailLevel](#) class has three public string properties that can be specified in the constructor, or set directly on the object. You then pass the [ODATADetailLevel](#) object as a parameter to the various list operations such as [ListPools](#), [ListJobs](#), and [ListTasks](#).

- [ODATADetailLevel.FilterClause](#): Limit the number of items that are returned.
- [ODATADetailLevel.SelectClause](#): Specify which property values are returned with each item.
- [ODATADetailLevel.ExpandClause](#): Retrieve data for all items in a single API call instead of separate calls for each item.

The following code snippet uses the Batch .NET API to efficiently query the Batch service for the statistics of a specific set of pools. In this scenario, the Batch user has both test and production pools. The test pool IDs are prefixed with "test", and the production pool IDs are prefixed with "prod". In the snippet, *myBatchClient* is a properly initialized instance of the [BatchClient](#) class.

```
// First we need an ODATADetailLevel instance on which to set the filter, select,
// and expand clause strings
ODATADetailLevel detailLevel = new ODATADetailLevel();

// We want to pull only the "test" pools, so we limit the number of items returned
// by using a FilterClause and specifying that the pool IDs must start with "test"
detailLevel.FilterClause = "startswith(id, 'test')";

// To further limit the data that crosses the wire, configure the SelectClause to
// limit the properties that are returned on each CloudPool object to only
// CloudPool.Id and CloudPool.Statistics
detailLevel.SelectClause = "id, stats";

// Specify the ExpandClause so that the .NET API pulls the statistics for the
// CloudPools in a single underlying REST API call. Note that we use the pool's
// REST API element name "stats" here as opposed to "Statistics" as it appears in
// the .NET API (CloudPool.Statistics)
detailLevel.ExpandClause = "stats";

// Now get our collection of pools, minimizing the amount of data that is returned
// by specifying the detail level that we configured above
List<CloudPool> testPools =
    await myBatchClient.PoolOperations.ListPools(detailLevel).ToListAsync();
```

TIP

An instance of [ODATADetailLevel](#) that is configured with Select and Expand clauses can also be passed to appropriate Get methods, such as [PoolOperations.GetPool](#), to limit the amount of data that is returned.

Batch REST to .NET API mappings

Property names in filter, select, and expand strings *must* reflect their REST API counterparts, both in name and case. The tables below provide mappings between the .NET and REST API counterparts.

Mappings for filter strings

- **.NET list methods:** Each of the .NET API methods in this column accepts an [ODATADetailLevel](#) object as a parameter.
- **REST list requests:** Each REST API page linked to in this column contains a table that specifies the properties and operations that are allowed in *filter* strings. You will use these property names and operations when you construct an [ODATADetailLevel.FilterClause](#) string.

.NET LIST METHODS	REST LIST REQUESTS
CertificateOperations.ListCertificates	List the certificates in an account
CloudTask.ListNodeFiles	List the files associated with a task
JobOperations.ListJobPreparationAndReleaseTaskStatus	List the status of the job preparation and job release tasks for a job
JobOperations.ListJobs	List the jobs in an account
JobOperations.ListNodeFiles	List the files on a node
JobOperations.ListTasks	List the tasks associated with a job
JobScheduleOperations.ListJobSchedules	List the job schedules in an account
JobScheduleOperations.ListJobs	List the jobs associated with a job schedule
PoolOperations.ListComputeNodes	List the compute nodes in a pool
PoolOperations.ListPools	List the pools in an account

Mappings for select strings

- **Batch .NET types:** Batch .NET API types.
- **REST API entities:** Each page in this column contains one or more tables that list the REST API property names for the type. These property names are used when you construct *select* strings. You will use these same property names when you construct an [ODATADetailLevel.SelectClause](#) string.

BATCH .NET TYPES	REST API ENTITIES
Certificate	Get information about a certificate
CloudJob	Get information about a job
CloudJobSchedule	Get information about a job schedule
ComputeNode	Get information about a node
CloudPool	Get information about a pool
CloudTask	Get information about a task

Example: construct a filter string

When you construct a filter string for [ODATADetailLevel.FilterClause](#), consult the table above under "Mappings for

filter strings" to find the REST API documentation page that corresponds to the list operation that you wish to perform. You will find the filterable properties and their supported operators in the first multirow table on that page. If you wish to retrieve all tasks whose exit code was nonzero, for example, this row on [List the tasks associated with a job](#) specifies the applicable property string and allowable operators:

PROPERTY	OPERATIONS ALLOWED	TYPE
executionInfo/exitCode	eq, ge, gt, le , lt	Int

Thus, the filter string for listing all tasks with a nonzero exit code would be:

```
(executionInfo/exitCode lt 0) or (executionInfo/exitCode gt 0)
```

Example: construct a select string

To construct [ODATADetailLevel.SelectClause](#), consult the table above under "Mappings for select strings" and navigate to the REST API page that corresponds to the type of entity that you are listing. You will find the selectable properties and their supported operators in the first multirow table on that page. If you wish to retrieve only the ID and command line for each task in a list, for example, you will find these rows in the applicable table on [Get information about a task](#):

PROPERTY	TYPE	NOTES
id	String	The ID of the task.
commandLine	String	The command line of the task.

The select string for including only the ID and command line with each listed task would then be:

```
id, commandLine
```

Code samples

Efficient list queries code sample

Check out the [EfficientListQueries](#) sample project on GitHub to see how efficient list querying can affect performance in an application. This C# console application creates and adds a large number of tasks to a job. Then, it makes multiple calls to the [JobOperations.ListTasks](#) method and passes [ODATADetailLevel](#) objects that are configured with different property values to vary the amount of data to be returned. It produces output similar to the following:

```
Adding 5000 tasks to job jobEffQuery...
5000 tasks added in 00:00:47.3467587, hit ENTER to query tasks...

4943 tasks retrieved in 00:00:04.3408081 (ExpandClause:  | FilterClause: state eq 'active' | SelectClause:
id,state)
0 tasks retrieved in 00:00:00.2662920 (ExpandClause:  | FilterClause: state eq 'running' | SelectClause:
id,state)
59 tasks retrieved in 00:00:00.3337760 (ExpandClause:  | FilterClause: state eq 'completed' | SelectClause:
id,state)
5000 tasks retrieved in 00:00:04.1429881 (ExpandClause:  | FilterClause:  | SelectClause: id,state)
5000 tasks retrieved in 00:00:15.1016127 (ExpandClause:  | FilterClause:  | SelectClause:
id,state,environmentSettings)
5000 tasks retrieved in 00:00:17.0548145 (ExpandClause: stats | FilterClause:  | SelectClause: )

Sample complete, hit ENTER to continue...
```

As shown in the elapsed times, you can greatly lower query response times by limiting the properties and the number of items that are returned. You can find this and other sample projects in the [azure-batch-samples](#) repository on GitHub.

BatchMetrics library and code sample

In addition to the EfficientListQueries code sample above, you can find the [BatchMetrics](#) project in the [azure-batch-samples](#) GitHub repository. The BatchMetrics sample project demonstrates how to efficiently monitor Azure Batch job progress using the Batch API.

The [BatchMetrics](#) sample includes a .NET class library project which you can incorporate into your own projects, and a simple command-line program to exercise and demonstrate the use of the library.

The sample application within the project demonstrates the following operations:

1. Selecting specific attributes in order to download only the properties you need
2. Filtering on state transition times in order to download only changes since the last query

For example, the following method appears in the BatchMetrics library. It returns an `ODATADetailLevel` that specifies that only the `id` and `state` properties should be obtained for the entities that are queried. It also specifies that only entities whose state has changed since the specified `DateTime` parameter should be returned.

```
internal static ODATADetailLevel OnlyChangedAfter(DateTime time)
{
    return new ODATADetailLevel(
        selectClause: "id, state",
        filterClause: string.Format("stateTransitionTime gt DateTime'{0:o}'", time)
    );
}
```

Next steps

Parallel node tasks

[Maximize Azure Batch compute resource usage with concurrent node tasks](#) is another article related to Batch application performance. Some types of workloads can benefit from executing parallel tasks on larger--but fewer--compute nodes. Check out the [example scenario](#) in the article for details on such a scenario.

Batch Forum

The [Azure Batch Forum](#) on MSDN is a great place to discuss Batch and ask questions about the service. Head on over for helpful "sticky" posts, and post your questions as they arise while you build your Batch solutions.

Run job preparation and job release tasks on Batch compute nodes

2/27/2017 • 7 min to read • [Edit Online](#)

An Azure Batch job often requires some form of setup before its tasks are executed, and post-job maintenance when its tasks are completed. You might need to download common task input data to your compute nodes, or upload task output data to Azure Storage after the job completes. You can use **job preparation** and **job release** tasks to perform these operations.

What are job preparation and release tasks?

Before a job's tasks run, the job preparation task runs on all compute nodes scheduled to run at least one task. Once the job is completed, the job release task runs on each node in the pool that executed at least one task. As with normal Batch tasks, you can specify a command line to be invoked when a job preparation or release task is run.

Job preparation and release tasks offer familiar Batch task features such as file download ([resource files](#)), elevated execution, custom environment variables, maximum execution duration, retry count, and file retention time.

In the following sections, you'll learn how to use the [JobPreparationTask](#) and [JobReleaseTask](#) classes found in the [Batch .NET](#) library.

TIP

Job preparation and release tasks are especially helpful in "shared pool" environments, in which a pool of compute nodes persists between job runs and is used by many jobs.

When to use job preparation and release tasks

Job preparation and job release tasks are a good fit for the following situations:

Download common task data

Batch jobs often require a common set of data as input for the job's tasks. For example, in daily risk analysis calculations, market data is job-specific, yet common to all tasks in the job. This market data, often several gigabytes in size, should be downloaded to each compute node only once so that any task that runs on the node can use it. Use a **job preparation task** to download this data to each node before the execution of the job's other tasks.

Delete job and task output

In a "shared pool" environment, where a pool's compute nodes are not decommissioned between jobs, you may need to delete job data between runs. You might need to conserve disk space on the nodes, or satisfy your organization's security policies. Use a **job release task** to delete data that was downloaded by a job preparation task, or generated during task execution.

Log retention

You might want to keep a copy of log files that your tasks generate, or perhaps crash dump files that can be generated by failed applications. Use a **job release task** in such cases to compress and upload this data to an [Azure Storage](#) account.

TIP

Another way to persist logs and other job and task output data is to use the [Azure Batch File Conventions](#) library.

Job preparation task

Before execution of a job's tasks, Batch executes the job preparation task on each compute node that is scheduled to run a task. By default, the Batch service waits for the job preparation task to be completed before running the tasks scheduled to execute on the node. However, you can configure the service not to wait. If the node restarts, the job preparation task runs again, but you can also disable this behavior.

The job preparation task is executed only on nodes that are scheduled to run a task. This prevents the unnecessary execution of a preparation task in case a node is not assigned a task. This can occur when the number of tasks for a job is less than the number of nodes in a pool. It also applies when [concurrent task execution](#) is enabled, which leaves some nodes idle if the task count is lower than the total possible concurrent tasks. By not running the job preparation task on idle nodes, you can spend less money on data transfer charges.

NOTE

[JobPreparationTask](#) differs from [CloudPool.StartTask](#) in that [JobPreparationTask](#) executes at the start of each job, whereas [StartTask](#) executes only when a compute node first joins a pool or restarts.

Job release task

Once a job is marked as completed, the job release task is executed on each node in the pool that executed at least one task. You mark a job as completed by issuing a terminate request. The Batch service then sets the job state to *terminating*, terminates any active or running tasks associated with the job, and runs the job release task. The job then moves to the *completed* state.

NOTE

Job deletion also executes the job release task. However, if a job has already been terminated, the release task is not run a second time if the job is later deleted.

Job prep and release tasks with Batch .NET

To use a job preparation task, assign a [JobPreparationTask](#) object to your job's [CloudJob.JobPreparationTask](#) property. Similarly, initialize a [JobReleaseTask](#) and assign it to your job's [CloudJob.JobReleaseTask](#) property to set the job's release task.

In this code snippet, `myBatchClient` is an instance of [BatchClient](#), and `myPool` is an existing pool within the Batch account.

```
// Create the CloudJob for CloudPool "myPool"
CloudJob myJob =
    myBatchClient.JobOperations.CreateJob(
        "JobPrepReleaseSampleJob",
        new PoolInformation() { PoolId = "myPool" });

// Specify the command lines for the job preparation and release tasks
string jobPrepCmdLine =
    "cmd /c echo %AZ_BATCH_NODE_ID% > %AZ_BATCH_NODE_SHARED_DIR%\shared_file.txt";
string jobReleaseCmdLine =
    "cmd /c del %AZ_BATCH_NODE_SHARED_DIR%\shared_file.txt";

// Assign the job preparation task to the job
myJob.JobPreparationTask =
    new JobPreparationTask { CommandLine = jobPrepCmdLine };

// Assign the job release task to the job
myJob.JobReleaseTask =
    new JobPreparationTask { CommandLine = jobReleaseCmdLine };

await myJob.CommitAsync();
```

As mentioned earlier, the release task is executed when a job is terminated or deleted. Terminate a job with [JobOperations.TerminateJobAsync](#). Delete a job with [JobOperations.DeleteJobAsync](#). You typically terminate or delete a job when its tasks are completed, or when a timeout that you've defined has been reached.

```
// Terminate the job to mark it as Completed; this will initiate the
// Job Release Task on any node that executed job tasks. Note that the
// Job Release Task is also executed when a job is deleted, thus you
// need not call Terminate if you typically delete jobs after task completion.
await myBatchClient.JobOperations.TerminateJobAsy("JobPrepReleaseSampleJob");
```

Code sample on GitHub

To see job preparation and release tasks in action, check out the [JobPrepRelease](#) sample project on GitHub. This console application does the following:

1. Creates a pool with two "small" nodes.
2. Creates a job with job preparation, release, and standard tasks.
3. Runs the job preparation task, which first writes the node ID to a text file in a node's "shared" directory.
4. Runs a task on each node that writes its task ID to the same text file.
5. Once all tasks are completed (or the timeout is reached), prints the contents of each node's text file to the console.
6. When the job is completed, runs the job release task to delete the file from the node.
7. Prints the exit codes of the job preparation and release tasks for each node on which they executed.
8. Pauses execution to allow confirmation of job and/or pool deletion.

Output from the sample application is similar to the following:

```
Attempting to create pool: JobPrepReleaseSamplePool
Created pool JobPrepReleaseSamplePool with 2 small nodes
Checking for existing job JobPrepReleaseSampleJob...
Job JobPrepReleaseSampleJob not found, creating...
Submitting tasks and awaiting completion...
All tasks completed.

Contents of shared\job_prep_and_release.txt on tvn-2434664350_1-20160623t173951z:
-----
tvn-2434664350_1-20160623t173951z tasks:
    task001
    task004
    task005
    task006

Contents of shared\job_prep_and_release.txt on tvn-2434664350_2-20160623t173951z:
-----
tvn-2434664350_2-20160623t173951z tasks:
    task008
    task002
    task003
    task007

Waiting for job JobPrepReleaseSampleJob to reach state Completed
...

tvn-2434664350_1-20160623t173951z:
    Prep task exit code: 0
    Release task exit code: 0

tvn-2434664350_2-20160623t173951z:
    Prep task exit code: 0
    Release task exit code: 0

Delete job? [yes] no
yes
Delete pool? [yes] no
yes

Sample complete, hit ENTER to exit...
```

NOTE

Due to the variable creation and start time of nodes in a new pool (some nodes are ready for tasks before others), you may see different output. Specifically, because the tasks complete quickly, one of the pool's nodes may execute all of the job's tasks. If this occurs, you will notice that the job prep and release tasks do not exist for the node that executed no tasks.

Inspect job preparation and release tasks in the Azure portal

When you run the sample application, you can use the [Azure portal](#) to view the properties of the job and its tasks, or even download the shared text file that is modified by the job's tasks.

The screenshot below shows the **Preparation tasks blade** in the Azure portal after a run of the sample application. Navigate to the *JobPrepReleaseSampleJob* properties after your tasks have completed (but before deleting your job and pool) and click **Preparation tasks** or **Release tasks** to view their properties.

JobPrepReleaseSampleJob - Preparation tasks

Columns Refresh

Search (Ctrl+/)

Overview

GENERAL

Properties

Environment settings

Metadata

Tasks

Preparation tasks

Release tasks

MANAGE

Priority

Constraints

Pool Info

Properties

ID: jobpreparation

Command line: cmd /c echo %AZ_BATCH_NODE_ID% tasks: >%AZ_BATCH_NODE_ID%

Run elevated: false

Resource files: 0 files

Environment settings: 0 environment settings

Wait for success: true

Rerun on node reboot: true

Select a query

Advanced query

Filter by pool ID or node ID

POOL ID	NODE ID	EXIT CODE	SCHEDULE ST...	STATE	STARTED
JobPrepRele...	tvm-25750...	0	✓	Completed	Sep 14, 16:5... ..

Next steps

Application packages

In addition to the job preparation task, you can also use the [application packages](#) feature of Batch to prepare compute nodes for task execution. This feature is especially useful for deploying applications that do not require running an installer, applications that contain many (100+) files, or applications that require strict version control.

Installing applications and staging data

This MSDN forum post provides an overview of several methods of preparing your nodes for running tasks:

[Installing applications and staging data on Batch compute nodes](#)

Written by one of the Azure Batch team members, it discusses several techniques that you can use to deploy applications and data to compute nodes.

Provision Linux compute nodes in Batch pools

2/27/2017 • 10 min to read • [Edit Online](#)

You can use Azure Batch to run parallel compute workloads on both Linux and Windows virtual machines. This article details how to create pools of Linux compute nodes in the Batch service by using both the [Batch Python](#) and [Batch .NET](#) client libraries.

NOTE

[Application packages](#) are currently unsupported on Linux compute nodes.

Virtual machine configuration

When you create a pool of compute nodes in Batch, you have two options from which to select the node size and operating system: Cloud Services Configuration and Virtual Machine Configuration.

Cloud Services Configuration provides Windows compute nodes *only*. Available compute node sizes are listed in [Sizes for Cloud Services](#), and available operating systems are listed in the [Azure Guest OS releases and SDK compatibility matrix](#). When you create a pool that contains Azure Cloud Services nodes, you need to specify only the node size and its "OS family," which are found in the previously mentioned articles. For pools of Windows compute nodes, Cloud Services is most commonly used.

Virtual Machine Configuration provides both Linux and Windows images for compute nodes. Available compute node sizes are listed in [Sizes for virtual machines in Azure](#) (Linux) and [Sizes for virtual machines in Azure](#) (Windows). When you create a pool that contains Virtual Machine Configuration nodes, you must specify the size of the nodes, the virtual machine image reference, and the Batch node agent SKU to be installed on the nodes.

Virtual machine image reference

The Batch service uses [Virtual machine scale sets](#) to provide Linux compute nodes. The operating system images for these virtual machines are provided by the [Azure Marketplace](#). When you configure a virtual machine image reference, you specify the properties of a Marketplace virtual machine image. The following properties are required when you create a virtual machine image reference:

IMAGE REFERENCE PROPERTIES	EXAMPLE
Publisher	Canonical
Offer	UbuntuServer
SKU	14.04.4-LTS
Version	latest

TIP

You can learn more about these properties and how to list Marketplace images in [Navigate and select Linux virtual machine images in Azure with CLI or PowerShell](#). Note that not all Marketplace images are currently compatible with Batch. For more information, see [Node agent SKU](#).

Node agent SKU

The Batch node agent is a program that runs on each node in the pool and provides the command-and-control interface between the node and the Batch service. There are different implementations of the node agent, known as SKUs, for different operating systems. Essentially, when you create a Virtual Machine Configuration, you first specify the virtual machine image reference, and then you specify the node agent to install on the image.

Typically, each node agent SKU is compatible with multiple virtual machine images. Here are a few examples of node agent SKUs:

- batch.node.ubuntu 14.04
- batch.node.centos 7
- batch.node.windows amd64

IMPORTANT

Not all virtual machine images that are available in the Marketplace are compatible with the currently available Batch node agents. You must use the Batch SDKs to list the available node agent SKUs and the virtual machine images with which they are compatible. See the [List of Virtual Machine images](#) later in this article for more information.

Create a Linux pool: Batch Python

The following code snippet shows an example of how to use the [Microsoft Azure Batch Client Library for Python](#) to create a pool of Ubuntu Server compute nodes. Reference documentation for the Batch Python module can be found at [azure.batch package](#) on Read the Docs.

This snippet creates an [ImageReference](#) explicitly and specifies each of its properties (publisher, offer, SKU, version). In production code, however, we recommend that you use the [list_node_agent_skus](#) method to determine and select from the available image and node agent SKU combinations at runtime.

```

# Import the required modules from the
# Azure Batch Client Library for Python
import azure.batch.batch_service_client as batch
import azure.batch.batch_auth as batchauth
import azure.batch.models as batchmodels

# Specify Batch account credentials
account = "<batch-account-name>"
key = "<batch-account-key>"
batch_url = "<batch-account-url>"

# Pool settings
pool_id = "LinuxNodesSamplePoolPython"
vm_size = "STANDARD_A1"
node_count = 1

# Initialize the Batch client
creds = batchauth.SharedKeyCredentials(account, key)
config = batch.BatchServiceClientConfiguration(creds, base_url = batch_url)
client = batch.BatchServiceClient(config)

# Create the unbound pool
new_pool = batchmodels.PoolAddParameter(id = pool_id, vm_size = vm_size)
new_pool.target_dedicated = node_count

# Configure the start task for the pool
start_task = batchmodels.StartTask()
start_task.run_elevated = True
start_task.command_line = "printenv AZ_BATCH_NODE_STARTUP_DIR"
new_pool.start_task = start_task

# Create an ImageReference which specifies the Marketplace
# virtual machine image to install on the nodes.
ir = batchmodels.ImageReference(
    publisher = "Canonical",
    offer = "UbuntuServer",
    sku = "14.04.2-LTS",
    version = "latest")

# Create the VirtualMachineConfiguration, specifying
# the VM image reference and the Batch node agent to
# be installed on the node.
vmc = batchmodels.VirtualMachineConfiguration(
    image_reference = ir,
    node_agent_sku_id = "batch.node.ubuntu 14.04")

# Assign the virtual machine configuration to the pool
new_pool.virtual_machine_configuration = vmc

# Create pool in the Batch service
client.pool.add(new_pool)

```

As mentioned previously, we recommend that instead of creating the [ImageReference](#) explicitly, you use the [list_node_agent_skus](#) method to dynamically select from the currently supported node agent/Marketplace image combinations. The following Python snippet shows usage of this method.


```
# Get the list of node agents from the Batch service
nodeagents = client.account.list_node_agent_skus()

# Obtain the desired node agent
ubuntu1404agent = next(agent for agent in nodeagents if "ubuntu 14.04" in agent.id)

# Pick the first image reference from the list of verified references
ir = ubuntu1404agent.verified_image_references[0]

# Create the VirtualMachineConfiguration, specifying the VM image
# reference and the Batch node agent to be installed on the node.
vmc = batchmodels.VirtualMachineConfiguration(
    image_reference = ir,
    node_agent_sku_id = ubuntu1404agent.id)
```

Create a Linux pool: Batch .NET

The following code snippet shows an example of how to use the [Batch .NET](#) client library to create a pool of Ubuntu Server compute nodes. You can find the [Batch .NET reference documentation](#) on MSDN.

The following code snippet uses the [PoolOperations.ListNodeAgentSkus](#) method to select from the list of currently supported Marketplace image and node agent SKU combinations. This technique is desirable because the list of supported combinations may change from time to time. Most commonly, supported combinations are added.

```
// Pool settings
const string poolId = "LinuxNodesSamplePoolDotNet";
const string vmSize = "STANDARD_A1";
const int nodeCount = 1;

// Obtain a collection of all available node agent SKUs.
// This allows us to select from a list of supported
// VM image/node agent combinations.
List<NodeAgentSku> nodeAgentSkus =
    batchClient.PoolOperations.ListNodeAgentSkus().ToList();

// Define a delegate specifying properties of the VM image
// that we wish to use.
Func<ImageReference, bool> isUbuntu1404 = imageRef =>
    imageRef.Publisher == "Canonical" &&
    imageRef.Offer == "UbuntuServer" &&
    imageRef.SkuId.Contains("14.04");

// Obtain the first node agent SKU in the collection that matches
// Ubuntu Server 14.04. Note that there are one or more image
// references associated with this node agent SKU.
NodeAgentSku ubuntuAgentSku = nodeAgentSkus.First(sku =>
    sku.VerifiedImageReferences.Any(isUbuntu1404));

// Select an ImageReference from those available for node agent.
ImageReference imageReference =
    ubuntuAgentSku.VerifiedImageReferences.First(isUbuntu1404);

// Create the VirtualMachineConfiguration for use when actually
// creating the pool
VirtualMachineConfiguration virtualMachineConfiguration =
    new VirtualMachineConfiguration(
        imageReference: imageReference,
        nodeAgentSkuId: ubuntuAgentSku.Id);

// Create the unbound pool object using the VirtualMachineConfiguration
// created above
CloudPool pool = batchClient.PoolOperations.CreatePool(
    poolId: poolId,
    virtualMachineSize: vmSize,
    virtualMachineConfiguration: virtualMachineConfiguration,
    targetDedicated: nodeCount);

// Commit the pool to the Batch service
pool.Commit();
```

Although the previous snippet uses the [PoolOperations.ListNodeAgentSkus](#) method to dynamically list and select from supported image and node agent SKU combinations (recommended), you can also configure an [ImageReference](#) explicitly:

```
ImageReference imageReference = new ImageReference(
    publisher: "Canonical",
    offer: "UbuntuServer",
    skuId: "14.04.2-LTS",
    version: "latest");
```

List of virtual machine images

The following table lists the Marketplace virtual machine images that are compatible with the available Batch node agents when this article was last updated. It is important to note that this list is not definitive because images and node agents may be added or removed at any time. We recommend that your Batch applications and services always use [list_node_agent_skus](#) (Python) and [ListNodeAgentSkus](#) (Batch .NET) to determine and select

from the currently available SKUs.

WARNING

The following list may change at any time. Always use the **list node agent SKU** methods available in the Batch APIs to list and then select from the compatible virtual machine and node agent SKUs when you run your Batch jobs.

PUBLISHER	OFFER	IMAGE SKU	VERSION	NODE AGENT SKU ID
Canonical	UbuntuServer	14.04.5-LTS	latest	batch.node.ubuntu 14.04
Canonical	UbuntuServer	16.04.0-LTS	latest	batch.node.ubuntu 16.04
Credativ	Debian	8	latest	batch.node.debian 8
OpenLogic	CentOS	7.0	latest	batch.node.centos 7
OpenLogic	CentOS	7.1	latest	batch.node.centos 7
OpenLogic	CentOS-HPC	7.1	latest	batch.node.centos 7
OpenLogic	CentOS	7.2	latest	batch.node.centos 7
Oracle	Oracle-Linux	7.0	latest	batch.node.centos 7
Oracle	Oracle-Linux	7.2	latest	batch.node.centos 7
SUSE	openSUSE	13.2	latest	batch.node.opensuse 13.2
SUSE	openSUSE-Leap	42.1	latest	batch.node.opensuse 42.1
SUSE	SLES	12-SP1	latest	batch.node.opensuse 42.1
SUSE	SLES-HPC	12-SP1	latest	batch.node.opensuse 42.1
microsoft-ads	linux-data-science-vm	linuxdsvm	latest	batch.node.centos 7
microsoft-ads	standard-data-science-vm	standard-data-science-vm	latest	batch.node.windows amd64
MicrosoftWindowsServer	WindowsServer	2008-R2-SP1	latest	batch.node.windows amd64
MicrosoftWindowsServer	WindowsServer	2012-Datacenter	latest	batch.node.windows amd64

PUBLISHER	OFFER	IMAGE SKU	VERSION	NODE AGENT SKU ID
MicrosoftWindowsServer	WindowsServer	2012-R2-Datacenter	latest	batch.node.windowsamd64
MicrosoftWindowsServer	WindowsServer	2016-Datacenter	latest	batch.node.windowsamd64
MicrosoftWindowsServer	WindowsServer	2016-Datacenter-with-Containers	latest	batch.node.windowsamd64

Connect to Linux nodes

During development or while troubleshooting, you may find it necessary to sign in to the nodes in your pool. Unlike Windows compute nodes, you cannot use Remote Desktop Protocol (RDP) to connect to Linux nodes. Instead, the Batch service enables SSH access on each node for remote connection.

The following Python code snippet creates a user on each node in a pool, which is required for remote connection. It then prints the secure shell (SSH) connection information for each node.

```

import datetime
import getpass
import azure.batch.batch_service_client as batch
import azure.batch.batch_auth as batchauth
import azure.batch.models as batchmodels

# Specify your own account credentials
batch_account_name = ''
batch_account_key = ''
batch_account_url = ''

# Specify the ID of an existing pool containing Linux nodes
# currently in the 'idle' state
pool_id = ''

# Specify the username and prompt for a password
username = 'linuxuser'
password = getpass.getpass()

# Create a BatchClient
credentials = batchauth.SharedKeyCredentials(
    batch_account_name,
    batch_account_key
)
batch_client = batch.BatchServiceClient(
    credentials,
    base_url=batch_account_url
)

# Create the user that will be added to each node in the pool
user = batchmodels.ComputeNodeUser(username)
user.password = password
user.is_admin = True
user.expiry_time = \
    (datetime.datetime.today() + datetime.timedelta(days=30)).isoformat()

# Get the list of nodes in the pool
nodes = batch_client.compute_node.list(pool_id)

# Add the user to each node in the pool and print
# the connection information for the node
for node in nodes:
    # Add the user to the node
    batch_client.compute_node.add_user(pool_id, node.id, user)

    # Obtain SSH login information for the node
    login = batch_client.compute_node.get_remote_login_settings(pool_id,
                                                                node.id)

    # Print the connection info for the node
    print("{0} | {1} | {2} | {3}".format(node.id,
                                         node.state,
                                         login.remote_login_ip_address,
                                         login.remote_login_port))

```

Here is sample output for the previous code for a pool that contains four Linux nodes:

```

Password:
tvm-1219235766_1-20160414t192511z | ComputeNodeState.idle | 13.91.7.57 | 50000
tvm-1219235766_2-20160414t192511z | ComputeNodeState.idle | 13.91.7.57 | 50003
tvm-1219235766_3-20160414t192511z | ComputeNodeState.idle | 13.91.7.57 | 50002
tvm-1219235766_4-20160414t192511z | ComputeNodeState.idle | 13.91.7.57 | 50001

```

Note that instead of a password, you can specify an SSH public key when you create a user on a node. In the

Python SDK, this is done by using the **ssh_public_key** parameter on [ComputeNodeUser](#). In .NET, this is done by using the [ComputeNodeUser.SshPublicKey](#) property.

Pricing

Azure Batch is built on Azure Cloud Services and Azure Virtual Machines technology. The Batch service itself is offered at no cost, which means you are charged only for the compute resources that your Batch solutions consume. When you choose **Cloud Services Configuration**, you will be charged based on the [Cloud Services pricing](#) structure. When you choose **Virtual Machine Configuration**, you will be charged based on the [Virtual Machines pricing](#) structure.

Next steps

Batch Python tutorial

For a more in-depth tutorial about how to work with Batch by using Python, check out [Get started with the Azure Batch Python client](#). Its companion [code sample](#) includes a helper function, `get_vm_config_for_distro`, that shows another technique to obtain a virtual machine configuration.

Batch Python code samples

Check out the other [Python code samples](#) in the [azure-batch-samples](#) repository on GitHub for several scripts that show you how to perform common Batch operations such as pool, job, and task creation. The [README](#) that accompanies the Python samples has details about how to install the required packages.

Batch forum

The [Azure Batch Forum](#) on MSDN is a great place to discuss Batch and ask questions about the service. Read helpful "stickied" posts, and post your questions as they arise while you build your Batch solutions.

Manage Batch accounts and quotas with the Batch Management client library for .NET

3/17/2017 • 6 min to read • [Edit Online](#)

You can lower maintenance overhead in your Azure Batch applications by using the [Batch Management .NET](#) library to automate Batch account creation, deletion, key management, and quota discovery.

- **Create and delete Batch accounts** within any region. If, as an independent software vendor (ISV) for example, you provide a service for your clients in which each is assigned a separate Batch account for billing purposes, you can add account creation and deletion capabilities to your customer portal.
- **Retrieve and regenerate account keys** programmatically for any of your Batch accounts. This can help you comply with security policies that enforce periodic rollover or expiry of account keys. When you have several Batch accounts in various Azure regions, automation of this rollover process increases your solution's efficiency.
- **Check account quotas** and take the trial-and-error guesswork out of determining which Batch accounts have what limits. By checking your account quotas before starting jobs, creating pools, or adding compute nodes, you can proactively adjust where or when these compute resources are created. You can determine which accounts require quota increases before allocating additional resources in those accounts.
- **Combine features of other Azure services** for a full-featured management experience--by using Batch Management .NET, [Azure Active Directory](#), and the [Azure Resource Manager](#) together in the same application. By using these features and their APIs, you can provide a frictionless authentication experience, the ability to create and delete resource groups, and the capabilities that are described above for an end-to-end management solution.

NOTE

While this article focuses on the programmatic management of your Batch accounts, keys, and quotas, you can perform many of these activities by using the [Azure portal](#). For more information, see [Create an Azure Batch account using the Azure portal](#) and [Quotas and limits for the Azure Batch service](#).

Create and delete Batch accounts

As mentioned, one of the primary features of the Batch Management API is to create and delete Batch accounts in an Azure region. To do so, use [BatchManagementClient.Account.CreateAsync](#) and [DeleteAsync](#), or their synchronous counterparts.

The following code snippet creates an account, obtains the newly created account from the Batch service, and then deletes it. In this snippet and the others in this article, `batchManagementClient` is a fully initialized instance of [BatchManagementClient](#).

```
// Create a new Batch account
await batchManagementClient.Account.CreateAsync("MyResourceGroup",
    "mynewaccount",
    new BatchAccountCreateParameters() { Location = "West US" });

// Get the new account from the Batch service
AccountResource account = await batchManagementClient.Account.GetAsync(
    "MyResourceGroup",
    "mynewaccount");

// Delete the account
await batchManagementClient.Account.DeleteAsync("MyResourceGroup", account.Name);
```

NOTE

Applications that use the Batch Management .NET library and its `BatchManagementClient` class require **service administrator** or **coadministrator** access to the subscription that owns the Batch account to be managed. For more information, see the [Azure Active Directory](#) section and the [AccountManagement](#) code sample.

Retrieve and regenerate account keys

Obtain primary and secondary account keys from any Batch account within your subscription by using [ListKeysAsync](#). You can regenerate those keys by using [RegenerateKeyAsync](#).

```
// Get and print the primary and secondary keys
BatchAccountListKeyResult accountKeys =
    await batchManagementClient.Account.ListKeysAsync(
        "MyResourceGroup",
        "mybatchaccount");
Console.WriteLine("Primary key: {0}", accountKeys.Primary);
Console.WriteLine("Secondary key: {0}", accountKeys.Secondary);

// Regenerate the primary key
BatchAccountRegenerateKeyResponse newKeys =
    await batchManagementClient.Account.RegenerateKeyAsync(
        "MyResourceGroup",
        "mybatchaccount",
        new BatchAccountRegenerateKeyParameters() {
            KeyName = AccountKeyType.Primary
        });
```

TIP

You can create a streamlined connection workflow for your management applications. First, obtain an account key for the Batch account you wish to manage with [ListKeysAsync](#). Then, use this key when initializing the Batch .NET library's [BatchSharedKeyCredentials](#) class, which is used when initializing [BatchClient](#).

Check Azure subscription and Batch account quotas

Azure subscriptions and the individual Azure services like Batch all have default quotas that limit the number of certain entities within them. For the default quotas for Azure subscriptions, see [Azure subscription and service limits, quotas, and constraints](#). For the default quotas of the Batch service, see [Quotas and limits for the Azure Batch service](#). By using the Batch Management .NET library, you can check these quotas in your applications. This enables you to make allocation decisions before you add accounts or compute resources like pools and compute nodes.

Check an Azure subscription for Batch account quotas

Before creating a Batch account in a region, you can check your Azure subscription to see whether you are able to add an account in that region.

In the code snippet below, we first use [BatchManagementClient.Account.ListAsync](#) to get a collection of all Batch accounts that are within a subscription. Once we've obtained this collection, we determine how many accounts are in the target region. Then we use [BatchManagementClient.Subscriptions](#) to obtain the Batch account quota and determine how many accounts (if any) can be created in that region.

```
// Get a collection of all Batch accounts within the subscription
BatchAccountListResponse listResponse =
    await batchManagementClient.Account.ListAsync(new AccountListParameters());
IList<AccountResource> accounts = listResponse.Accounts;
Console.WriteLine("Total number of Batch accounts under subscription id {0}: {1}",
    creds.SubscriptionId,
    accounts.Count);

// Get a count of all accounts within the target region
string region = "westus";
int accountsInRegion = accounts.Count(o => o.Location == region);

// Get the account quota for the specified region
SubscriptionQuotasGetResponse quotaResponse = await
batchManagementClient.Subscriptions.GetSubscriptionQuotasAsync(region);
Console.WriteLine("Account quota for {0} region: {1}", region, quotaResponse.AccountQuota);

// Determine how many accounts can be created in the target region
Console.WriteLine("Accounts in {0}: {1}", region, accountsInRegion);
Console.WriteLine("You can create {0} accounts in the {1} region.", quotaResponse.AccountQuota -
accountsInRegion, region);
```

In the snippet above, `creds` is an instance of [TokenCloudCredentials](#). To see an example of creating this object, see the [AccountManagement](#) code sample on GitHub.

Check a Batch account for compute resource quotas

Before increasing compute resources in your Batch solution, you can check to ensure the resources you want to allocate won't exceed the account's quotas. In the code snippet below, we print the quota information for the Batch account named `mybatchaccount`. In your own application, you could use such information to determine whether the account can handle the additional resources to be created.

```
// First obtain the Batch account
BatchAccountGetResponse getResponse =
    await batchManagementClient.Account.GetAsync("MyResourceGroup", "mybatchaccount");
AccountResource account = getResponse.Resource;

// Now print the compute resource quotas for the account
Console.WriteLine("Core quota: {0}", account.Properties.CoreQuota);
Console.WriteLine("Pool quota: {0}", account.Properties.PoolQuota);
Console.WriteLine("Active job and job schedule quota: {0}", account.Properties.ActiveJobAndJobScheduleQuota);
```

IMPORTANT

While there are default quotas for Azure subscriptions and services, many of these limits can be raised by issuing a request in the [Azure portal](#). For example, see [Quotas and limits for the Azure Batch service](#) for instructions on increasing your Batch account quotas.

Use Azure AD with Batch Management .NET

The Batch Management .NET library is an Azure resource provider client, and is used together with [Azure Resource Manager](#) to manage account resources programmatically. Azure AD is required to authenticate requests made through any Azure resource provider client, including the Batch Management .NET library, and through [Azure Resource Manager](#). For information about using Azure AD with the Batch Management .NET library, see [Use Azure Active Directory to authenticate Batch solutions](#).

Sample project on GitHub

To see Batch Management .NET in action, check out the [AccountManagment](#) sample project on GitHub. The AccountManagment sample application demonstrates the following operations:

1. Acquire a security token from Azure AD by using [ADAL](#). If the user is not already signed in, they are prompted for their Azure credentials.
2. With the security token obtained from Azure AD, create a [SubscriptionClient](#) to query Azure for a list of subscriptions associated with the account. The user can select a subscription from the list if it contains more than one subscription.
3. Get credentials associated with the selected subscription.
4. Create a [ResourceManagementClient](#) object by using the credentials.
5. Use a [ResourceManagementClient](#) object to create a resource group.
6. Use a [BatchManagementClient](#) object to perform several Batch account operations:
 - Create a Batch account in the new resource group.
 - Get the newly created account from the Batch service.
 - Print the account keys for the new account.
 - Regenerate a new primary key for the account.
 - Print the quota information for the account.
 - Print the quota information for the subscription.
 - Print all accounts within the subscription.
 - Delete newly created account.
7. Delete the resource group.

Before deleting the newly created Batch account and resource group, you can view them in the [Azure portal](#):

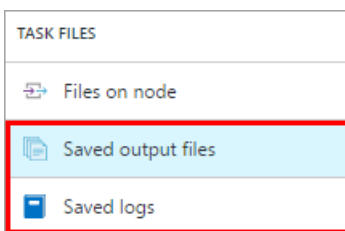
To run the sample application successfully, you must first register it with your Azure AD tenant in the Azure portal and grant permissions to the Azure Resource Manager API. Follow the steps provided in [Authenticate Batch management applications with Azure AD](#).

Persist results from completed jobs and tasks to Azure Storage

3/8/2017 • 11 min to read • [Edit Online](#)

The tasks you run in Batch typically produce output that must be stored and then later retrieved by other tasks in the job, the client application that executed the job, or both. This output might be files created by processing input data or log files associated with task execution. This article introduces a .NET class library that uses a conventions-based technique to persist such task output to Azure Blob storage, making it available even after you delete your pools, jobs, and compute nodes.

By using the technique in this article, you can also view your task output in **Saved output files** and **Saved logs** in the [Azure portal](#).



NOTE

The [Azure Batch File Conventions](#) .NET class library discussed in this article is currently in preview. Some of the features described here may change prior to general availability.

Task output considerations

When you design your Batch solution, you must consider several factors related to job and task outputs.

- **Compute node lifetime:** Compute nodes are often transient, especially in autoscale-enabled pools. The outputs of the tasks that run on a node are available only while the node exists, and only within the file retention time you've set for the task. To ensure that the task output is preserved, your tasks must therefore upload their output files to a durable store, for example, Azure Storage.
- **Output storage:** To persist task output data to durable storage, you can use the [Azure Storage SDK](#) in your task code to upload the task output to a Blob storage container. If you implement a container and file naming convention, your client application or other tasks in the job can then locate and download this output based on the convention.
- **Output retrieval:** You can retrieve task output directly from the compute nodes in your pool, or from Azure Storage if your tasks persist their output. To retrieve a task's output directly from a compute node, you need the file name and its output location on the node. If you persist output to Azure Storage, downstream tasks or your client application must have the full path to the file in Azure Storage to download it by using the Azure Storage SDK.
- **Viewing output:** When you navigate to a Batch task in the Azure portal and select **Files on node**, you are presented with all files associated with the task, not just the output files you're interested in. Again, files on compute nodes are available only while the node exists and only within the file retention time you've set for the task. To view task output that you've persisted to Azure Storage in the portal or an application like the [Azure Storage Explorer](#), you must know its location and navigate to the file directly.

Help for persisted output

To help you more easily persist job and task output, the Batch team has defined and implemented a set of naming conventions as well as a .NET class library, the [Azure Batch File Conventions](#) library, that you can use in your Batch applications. In addition, the Azure portal is aware of these naming conventions so that you can easily find the files you've stored by using the library.

Using the file conventions library

[Azure Batch File Conventions](#) is a .NET class library that your Batch .NET applications can use to easily store and retrieve task outputs to and from Azure Storage. It is intended for use in both task and client code--in task code for persisting files, and in client code to list and retrieve them. Your tasks can also use the library for retrieving the outputs of upstream tasks, such as in a [task dependencies](#) scenario.

The conventions library takes care of ensuring that storage containers and task output files are named according to the convention, and are uploaded to the right place when persisted to Azure Storage. When you retrieve outputs, you can easily locate the outputs for a given job or task by listing or retrieving the outputs by ID and purpose, instead of having to know filenames or where it exists in Storage.

For example, you can use the library to "list all intermediate files for task 7," or "get me the thumbnail preview for job *mymovie*," without needing to know the file names or location within your Storage account.

Get the library

You can obtain the library, which contains new classes and extends the [CloudJob](#) and [CloudTask](#) classes with new methods, from [NuGet](#). You can add it to your Visual Studio project using the [NuGet Library Package Manager](#).

TIP

You can find the [source code](#) for the Azure Batch File Conventions library on GitHub in the Microsoft Azure SDK for .NET repository.

Requirement: linked storage account

To store outputs to durable storage using the file conventions library and view them in the Azure portal, you must [link an Azure Storage account](#) to your Batch account. If you haven't already, link a Storage account to your Batch account by using the Azure portal:

Batch account blade > **Settings** > **Storage Account** > **Storage Account** (None) > Select a Storage account in your subscription

For a more detailed walk-through on linking a Storage account, see [Application deployment with Azure Batch application packages](#).

Persist output

There are two primary actions to perform when saving job and task output with the file conventions library: create the storage container and save output to the container.

WARNING

Because all job and task outputs are stored in the same container, [storage throttling limits](#) may be enforced if a large number of tasks try to persist files at the same time.

Create storage container

Before your tasks begin persisting output to storage, you must create a blob storage container to which they'll upload their output. Do this by calling [CloudJob.PrepareOutputStorageAsync](#). This extension method takes a [CloudStorageAccount](#) object as a parameter, and creates a container named in such a way that its contents are discoverable by the Azure portal and the retrieval methods discussed later in the article.

You typically place this code in your client application--the application that creates your pools, jobs, and tasks.

```
CloudJob job = batchClient.JobOperations.CreateJob(
    "myJob",
    new PoolInformation { PoolId = "myPool" });

// Create reference to the linked Azure Storage account
CloudStorageAccount linkedStorageAccount =
    new CloudStorageAccount(myCredentials, true);

// Create the blob storage container for the outputs
await job.PrepareOutputStorageAsync(linkedStorageAccount);
```

Store task outputs

Now that you've prepared a container in blob storage, tasks can save output to the container by using the [TaskOutputStorage](#) class found in the file conventions library.

In your task code, first create a [TaskOutputStorage](#) object, then when the task has completed its work, call the [TaskOutputStorage.SaveAsync](#) method to save its output to Azure Storage.

```
CloudStorageAccount linkedStorageAccount = new CloudStorageAccount(myCredentials);
string jobId = Environment.GetEnvironmentVariable("AZ_BATCH_JOB_ID");
string taskId = Environment.GetEnvironmentVariable("AZ_BATCH_TASK_ID");

TaskOutputStorage taskOutputStorage = new TaskOutputStorage(
    linkedStorageAccount, jobId, taskId);

/* Code to process data and produce output file(s) */

await taskOutputStorage.SaveAsync(TaskOutputKind.TaskOutput, "frame_full_res.jpg");
await taskOutputStorage.SaveAsync(TaskOutputKind.TaskPreview, "frame_low_res.jpg");
```

The "output kind" parameter categorizes the persisted files. There are four predefined [TaskOutputKind](#) types: "TaskOutput", "TaskPreview", "TaskLog", and "TaskIntermediate." You can also define custom kinds if they would be useful in your workflow.

These output types allow you to specify which type of outputs to list when you later query Batch for the persisted outputs of a given task. In other words, when you list the outputs for a task, you can filter the list on one of the output types. For example, "Give me the *preview* output for task 109." More on listing and retrieving outputs appears in [Retrieve output](#) later in the article.

TIP

The output kind also designates where in the Azure portal a particular file appears: *TaskOutput*-categorized files appear in "Task output files", and *TaskLog* files appear in "Task logs."

Store job outputs

In addition to storing task outputs, you can store the outputs associated with an entire job. For example, in the merge task of a movie rendering job, you could persist the fully rendered movie as a job output. When your job is completed, your client application can simply list and retrieve the outputs for the job, and does not need to query the individual tasks.

Store job output by calling the [JobOutputStorage.SaveAsync](#) method, and specify the [JobOutputKind](#) and filename:

```
CloudJob job = await batchClient.JobOperations.GetJobAsync(jobId);
JobOutputStorage jobOutputStorage = job.OutputStorage(linkedStorageAccount);

await jobOutputStorage.SaveAsync(JobOutputKind.JobOutput, "mymovie.mp4");
await jobOutputStorage.SaveAsync(JobOutputKind.JobPreview, "mymovie_preview.mp4");
```

As with [TaskOutputKind](#) for task outputs, you use the [JobOutputKind](#) parameter to categorize a job's persisted files. This parameter allows you to later query for (list) a specific type of output. The [JobOutputKind](#) includes both output and preview types, and supports creating custom types.

Store task logs

In addition to persisting a file to durable storage when a task or job completes, you might find it necessary to persist files that are updated during the execution of a task--log files or `stdout.txt` and `stderr.txt`, for example. For this purpose, the Azure Batch File Conventions library provides the [TaskOutputStorage.SaveTrackedAsync](#) method. With [SaveTrackedAsync](#), you can track updates to a file on the node (at an interval that you specify) and persist those updates to Azure Storage.

In the following code snippet, we use [SaveTrackedAsync](#) to update `stdout.txt` in Azure Storage every 15 seconds during the execution of the task:

```
TimeSpan stdoutFlushDelay = TimeSpan.FromSeconds(3);
string logFilePath = Path.Combine(
    Environment.GetEnvironmentVariable("AZ_BATCH_TASK_DIR"), "stdout.txt");

// The primary task logic is wrapped in a using statement that sends updates to
// the stdout.txt blob in Storage every 15 seconds while the task code runs.
using (ITrackedSaveOperation stdout =
    await taskStorage.SaveTrackedAsync(
        TaskOutputKind.TaskLog,
        logFilePath,
        "stdout.txt",
        TimeSpan.FromSeconds(15)))
{
    /* Code to process data and produce output file(s) */

    // We are tracking the disk file to save our standard output, but the
    // node agent may take up to 3 seconds to flush the stdout stream to
    // disk. So give the file a moment to catch up.
    await Task.Delay(stdoutFlushDelay);
}
```

`Code to process data and produce output file(s)` is simply a placeholder for the code that your task would normally perform. For example, you might have code that downloads data from Azure Storage and performs transformation or calculation on it. The important part of this snippet is demonstrating how you can wrap such code in a `using` block to periodically update a file with [SaveTrackedAsync](#).

The `Task.Delay` is required at the end of this `using` block to ensure that the node agent has time to flush the contents of standard out to the `stdout.txt` file on the node (the node agent is a program that runs on each node in the pool and provides the command-and-control interface between the node and the Batch service). Without this delay, it is possible to miss the last few seconds of output. This delay may not be required for all files.

NOTE

When you enable file tracking with `SaveTrackedAsync`, only *appends* to the tracked file are persisted to Azure Storage. Use this method only for tracking non-rotating log files or other files that are appended to, that is, data is only added to the end of the file when it's updated.

Retrieve output

When you retrieve your persisted output using the Azure Batch File Conventions library, you do so in a task- and job-centric manner. You can request the output for given task or job without needing to know its path in blob Storage, or even its file name. You can simply say, "Give me the output files for task *109*."

The following code snippet iterates through all of a job's tasks, prints some information about the output files for the task, and then downloads its files from Storage.

```
foreach (CloudTask task in myJob.ListTasks())
{
    foreach (TaskOutputStorage output in
        task.OutputStorage(storageAccount).ListOutputs(
            TaskOutputKind.TaskOutput))
    {
        Console.WriteLine($"output file: {output.FilePath}");

        output.DownloadToFileAsync(
            $"{jobId}-{output.FilePath}",
            System.IO.FileMode.Create).Wait();
    }
}
```

Task outputs and the Azure portal

The Azure portal displays task outputs and logs that are persisted to a linked Azure Storage account using the naming conventions found in the [Azure Batch File Conventions README](#). You can implement these conventions yourself in a language of your choosing, or you can use the file conventions library in your .NET applications.

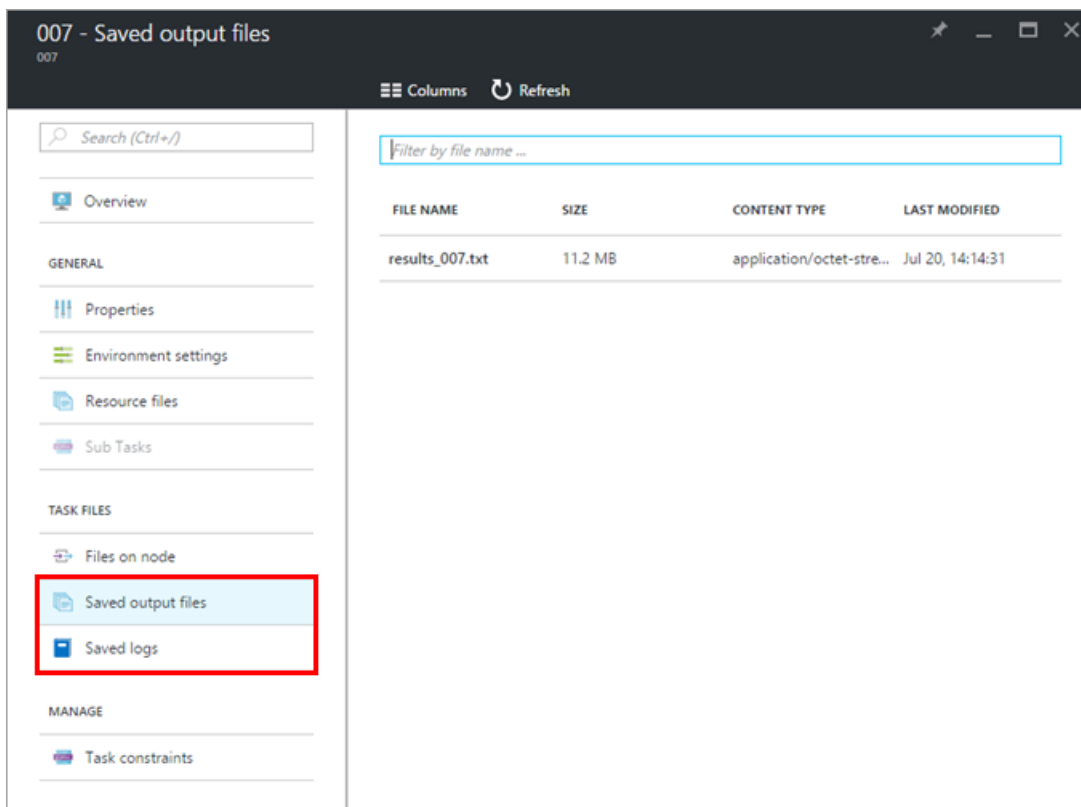
Enable portal display

To enable the display of your outputs in the portal, you must satisfy the following requirements:

1. [Link an Azure Storage account](#) to your Batch account.
2. Adhere to the predefined naming conventions for Storage containers and files when persisting outputs. You can find the definition of these conventions in the file conventions library [README](#). If you use the [Azure Batch File Conventions](#) library to persist your output, this requirement is satisfied.

View outputs in the portal

To view task outputs and logs in the Azure portal, navigate to the task whose output you are interested in, then click either **Saved output files** or **Saved logs**. This image shows the **Saved output files** for the task with ID "007":



Code sample

The [PersistOutputs](#) sample project is one of the [Azure Batch code samples](#) on GitHub. This Visual Studio solution demonstrates how to use the Azure Batch File Conventions library to persist task output to durable storage. To run the sample, follow these steps:

1. Open the project in **Visual Studio 2015 or newer**.
2. Add your Batch and Storage **account credentials** to **AccountSettings.settings** in the `Microsoft.Azure.Batch.Samples.Common` project.
3. **Build** (but do not run) the solution. Restore any NuGet packages if prompted.
4. Use the Azure portal to upload an [application package](#) for **PersistOutputsTask**. Include the `PersistOutputsTask.exe` and its dependent assemblies in the .zip package, set the application ID to "PersistOutputsTask", and the application package version to "1.0".
5. **Start** (run) the **PersistOutputs** project.

Next steps

Application deployment

The [application packages](#) feature of Batch provides an easy way to both deploy and version the applications that your tasks execute on compute nodes.

Installing applications and staging data

Check out the [Installing applications and staging data on Batch compute nodes](#) post in the Azure Batch forum for an overview of the various methods of preparing your nodes for running tasks. Written by one of the Azure Batch team members, this post is a good primer on the different ways to get files (including both applications and task input data) onto your compute nodes, and some special considerations for each method.

Use multi-instance tasks to run Message Passing Interface (MPI) applications in Batch

3/28/2017 • 13 min to read • [Edit Online](#)

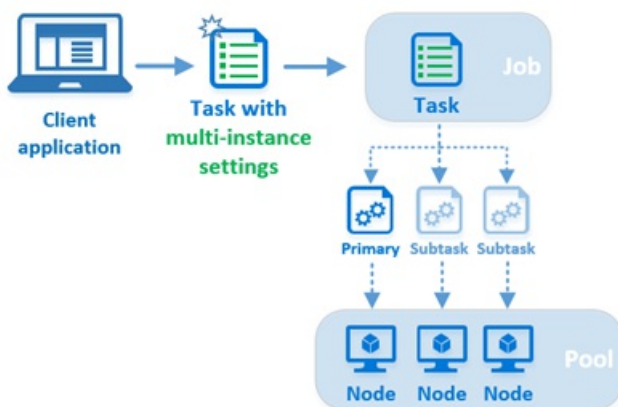
Multi-instance tasks allow you to run an Azure Batch task on multiple compute nodes simultaneously. These tasks enable high performance computing scenarios like Message Passing Interface (MPI) applications in Batch. In this article, you learn how to execute multi-instance tasks using the [Batch .NET](#) library.

NOTE

While the examples in this article focus on Batch .NET, MS-MPI, and Windows compute nodes, the multi-instance task concepts discussed here are applicable to other platforms and technologies (Python and Intel MPI on Linux nodes, for example).

Multi-instance task overview

In Batch, each task is normally executed on a single compute node--you submit multiple tasks to a job, and the Batch service schedules each task for execution on a node. However, by configuring a task's **multi-instance settings**, you tell Batch to instead create one primary task and several subtasks that are then executed on multiple nodes.



When you submit a task with multi-instance settings to a job, Batch performs several steps unique to multi-instance tasks:

1. The Batch service creates one **primary** and several **subtasks** based on the multi-instance settings. The total number of tasks (primary plus all subtasks) matches the number of **instances** (compute nodes) you specify in the multi-instance settings.
2. Batch designates one of the compute nodes as the **master**, and schedules the primary task to execute on the master. It schedules the subtasks to execute on the remainder of the compute nodes allocated to the multi-instance task, one subtask per node.
3. The primary and all subtasks download any **common resource files** you specify in the multi-instance settings.
4. After the common resource files have been downloaded, the primary and subtasks execute the **coordination command** you specify in the multi-instance settings. The coordination command is typically used to prepare nodes for executing the task. This can include starting background services (such as [Microsoft MPI's](#) `smpd.exe`) and verifying that the nodes are ready to process inter-node messages.
5. The primary task executes the **application command** on the master node *after* the coordination command

has been completed successfully by the primary and all subtasks. The application command is the command line of the multi-instance task itself, and is executed only by the primary task. In an [MS-MPI](#)-based solution, this is where you execute your MPI-enabled application using `mpiexec.exe`.

NOTE

Though it is functionally distinct, the "multi-instance task" is not a unique task type like the [StartTask](#) or [JobPreparationTask](#). The multi-instance task is simply a standard Batch task ([CloudTask](#) in Batch .NET) whose multi-instance settings have been configured. In this article, we refer to this as the **multi-instance task**.

Requirements for multi-instance tasks

Multi-instance tasks require a pool with **inter-node communication enabled**, and with **concurrent task execution disabled**. If you try to run a multi-instance task in a pool with internode communication disabled, or with a `maxTasksPerNode` value greater than 1, the task is never scheduled--it remains indefinitely in the "active" state. This code snippet shows the creation of such a pool using the Batch .NET library.

```
CloudPool myCloudPool =
    myBatchClient.PoolOperations.CreatePool(
        poolId: "MultiInstanceSamplePool",
        targetDedicated: 3
        virtualMachineSize: "small",
        cloudServiceConfiguration: new CloudServiceConfiguration(osFamily: "4"));

// Multi-instance tasks require inter-node communication, and those nodes
// must run only one task at a time.
myCloudPool.InterComputeNodeCommunicationEnabled = true;
myCloudPool.MaxTasksPerComputeNode = 1;
```

Additionally, multi-instance tasks can execute *only* on nodes in **pools created after 14 December 2015**.

Use a StartTask to install MPI

To run MPI applications with a multi-instance task, you first need to install an MPI implementation (MS-MPI or Intel MPI, for example) on the compute nodes in the pool. This is a good time to use a [StartTask](#), which executes whenever a node joins a pool, or is restarted. This code snippet creates a StartTask that specifies the MS-MPI setup package as a [resource file](#). The start task's command line is executed after the resource file is downloaded to the node. In this case, the command line performs an unattended install of MS-MPI.

```
// Create a StartTask for the pool which we use for installing MS-MPI on
// the nodes as they join the pool (or when they are restarted).
StartTask startTask = new StartTask
{
    CommandLine = "cmd /c MSMpiSetup.exe -unattend -force",
    ResourceFiles = new List<ResourceFile> { new
ResourceFile("https://mystorageaccount.blob.core.windows.net/mycontainer/MSMpiSetup.exe", "MSMpiSetup.exe") },
    UserIdentity = new UserIdentity(new AutoUserSpecification(elevationLevel: ElevationLevel.Admin)),
    WaitForSuccess = true
};
myCloudPool.StartTask = startTask;

// Commit the fully configured pool to the Batch service to actually create
// the pool and its compute nodes.
await myCloudPool.CommitAsync();
```

Remote direct memory access (RDMA)

When you choose an [RDMA-capable size](#) such as A9 for the compute nodes in your Batch pool, your MPI application can take advantage of Azure's high-performance, low-latency remote direct memory access (RDMA)

network.

Look for the sizes specified as "RDMA capable" in the following articles:

- **CloudServiceConfiguration** pools
 - [Sizes for Cloud Services](#) (Windows only)
- **VirtualMachineConfiguration** pools
 - [Sizes for virtual machines in Azure](#) (Linux)
 - [Sizes for virtual machines in Azure](#) (Windows)

NOTE

To take advantage of RDMA on [Linux compute nodes](#), you must use **Intel MPI** on the nodes. For more information on CloudServiceConfiguration and VirtualMachineConfiguration pools, see the Pool section of the [Batch feature overview](#).

Create a multi-instance task with Batch .NET

Now that we've covered the pool requirements and MPI package installation, let's create the multi-instance task. In this snippet, we create a standard [CloudTask](#), then configure its [MultiInstanceSettings](#) property. As mentioned earlier, the multi-instance task is not a distinct task type, but a standard Batch task configured with multi-instance settings.

```
// Create the multi-instance task. Its command line is the "application command"
// and will be executed *only* by the primary, and only after the primary and
// subtasks execute the CoordinationCommandLine.
CloudTask myMultiInstanceTask = new CloudTask(id: "mymultiinstancetask",
    commandline: "cmd /c mpiexec.exe -wdir %AZ_BATCH_TASK_SHARED_DIR% MyMPIApplication.exe");

// Configure the task's MultiInstanceSettings. The CoordinationCommandLine will be executed by
// the primary and all subtasks.
myMultiInstanceTask.MultiInstanceSettings =
    new MultiInstanceSettings(numberOfNodes) {
        CoordinationCommandLine = @"cmd /c start cmd /c ""%MSMPI_BIN%\smpd.exe"" -d",
        CommonResourceFiles = new List<ResourceFile> {
            new ResourceFile("https://mystorageaccount.blob.core.windows.net/mycontainer/MyMPIApplication.exe",
                "MyMPIApplication.exe")
        }
    };

// Submit the task to the job. Batch will take care of splitting it into subtasks and
// scheduling them for execution on the nodes.
await myBatchClient.JobOperations.AddTaskAsync("mybatchjob", myMultiInstanceTask);
```

Primary task and subtasks

When you create the multi-instance settings for a task, you specify the number of compute nodes that are to execute the task. When you submit the task to a job, the Batch service creates one **primary** task and enough **subtasks** that together match the number of nodes you specified.

These tasks are assigned an integer id in the range of 0 to *numberOfInstances* - 1. The task with id 0 is the primary task, and all other ids are subtasks. For example, if you create the following multi-instance settings for a task, the primary task would have an id of 0, and the subtasks would have ids 1 through 9.

```
int numberOfNodes = 10;
myMultiInstanceTask.MultiInstanceSettings = new MultiInstanceSettings(numberOfNodes);
```

Master node

When you submit a multi-instance task, the Batch service designates one of the compute nodes as the "master" node, and schedules the primary task to execute on the master node. The subtasks are scheduled to execute on the remainder of the nodes allocated to the multi-instance task.

Coordination command

The **coordination command** is executed by both the primary and subtasks.

The invocation of the coordination command is blocking--Batch does not execute the application command until the coordination command has returned successfully for all subtasks. The coordination command should therefore start any required background services, verify that they are ready for use, and then exit. For example, this coordination command for a solution using MS-MPI version 7 starts the SMPD service on the node, then exits:

```
cmd /c start cmd /c "%MSMPI_BIN%\smpd.exe" -d
```

Note the use of `start` in this coordination command. This is required because the `smpd.exe` application does not return immediately after execution. Without the use of the `start` command, this coordination command would not return, and would therefore block the application command from running.

Application command

Once the primary task and all subtasks have finished executing the coordination command, the multi-instance task's command line is executed by the primary task *only*. We call this the **application command** to distinguish it from the coordination command.

For MS-MPI applications, use the application command to execute your MPI-enabled application with `mpiexec.exe`. For example, here is an application command for a solution using MS-MPI version 7:

```
cmd /c "%MSMPI_BIN%\mpiexec.exe" -c 1 -wdir %AZ_BATCH_TASK_SHARED_DIR% MyMPIApplication.exe
```

NOTE

Because MS-MPI's `mpiexec.exe` uses the `CCP_NODES` variable by default (see [Environment variables](#)) the example application command line above excludes it.

Environment variables

Batch creates several [environment variables](#) specific to multi-instance tasks on the compute nodes allocated to a multi-instance task. Your coordination and application command lines can reference these environment variables, as can the scripts and programs they execute.

The following environment variables are created by the Batch service for use by multi-instance tasks:

- `CCP_NODES`
- `AZ_BATCH_NODE_LIST`
- `AZ_BATCH_HOST_LIST`
- `AZ_BATCH_MASTER_NODE`
- `AZ_BATCH_TASK_SHARED_DIR`
- `AZ_BATCH_IS_CURRENT_NODE_MASTER`

For full details on these and the other Batch compute node environment variables, including their contents and

visibility, see [Compute node environment variables](#).

TIP

The Batch Linux MPI code sample contains an example of how several of these environment variables can be used. The [coordination-cmd](#) Bash script downloads common application and input files from Azure Storage, enables a Network File System (NFS) share on the master node, and configures the other nodes allocated to the multi-instance task as NFS clients.

Resource files

There are two sets of resource files to consider for multi-instance tasks: **common resource files** that *all* tasks download (both primary and subtasks), and the **resource files** specified for the multi-instance task itself, which *only the primary* task downloads.

You can specify one or more **common resource files** in the multi-instance settings for a task. These common resource files are downloaded from [Azure Storage](#) into each node's **task shared directory** by the primary and all subtasks. You can access the task shared directory from application and coordination command lines by using the `AZ_BATCH_TASK_SHARED_DIR` environment variable. The `AZ_BATCH_TASK_SHARED_DIR` path is identical on every node allocated to the multi-instance task, thus you can share a single coordination command between the primary and all subtasks. Batch does not "share" the directory in a remote access sense, but you can use it as a mount or share point as mentioned earlier in the tip on environment variables.

Resource files that you specify for the multi-instance task itself are downloaded to the task's working directory, `AZ_BATCH_TASK_WORKING_DIR`, by default. As mentioned, in contrast to common resource files, only the primary task downloads resource files specified for the multi-instance task itself.

IMPORTANT

Always use the environment variables `AZ_BATCH_TASK_SHARED_DIR` and `AZ_BATCH_TASK_WORKING_DIR` to refer to these directories in your command lines. Do not attempt to construct the paths manually.

Task lifetime

The lifetime of the primary task controls the lifetime of the entire multi-instance task. When the primary exits, all of the subtasks are terminated. The exit code of the primary is the exit code of the task, and is therefore used to determine the success or failure of the task for retry purposes.

If any of the subtasks fail, exiting with a non-zero return code, for example, the entire multi-instance task fails. The multi-instance task is then terminated and retried, up to its retry limit.

When you delete a multi-instance task, the primary and all subtasks are also deleted by the Batch service. All subtask directories and their files are deleted from the compute nodes, just as for a standard task.

[TaskConstraints](#) for a multi-instance task, such as the [MaxTaskRetryCount](#), [MaxWallClockTime](#), and [RetentionTime](#) properties, are honored as they are for a standard task, and apply to the primary and all subtasks. However, if you change the [RetentionTime](#) property after adding the multi-instance task to the job, this change is applied only to the primary task. All of the subtasks continue to use the original [RetentionTime](#).

A compute node's recent task list reflects the id of a subtask if the recent task was part of a multi-instance task.

Obtain information about subtasks

To obtain information on subtasks by using the Batch .NET library, call the [CloudTask.ListSubtasks](#) method. This method returns information on all subtasks, and information about the compute node that executed the tasks.

From this information, you can determine each subtask's root directory, the pool id, its current state, exit code, and more. You can use this information in combination with the [PoolOperations.GetNodeFile](#) method to obtain the subtask's files. Note that this method does not return information for the primary task (id 0).

NOTE

Unless otherwise stated, Batch .NET methods that operate on the multi-instance [CloudTask](#) itself apply *only* to the primary task. For example, when you call the [CloudTask.ListNodeFiles](#) method on a multi-instance task, only the primary task's files are returned.

The following code snippet shows how to obtain subtask information, as well as request file contents from the nodes on which they executed.

```
// Obtain the job and the multi-instance task from the Batch service
CloudJob boundJob = batchClient.JobOperations.GetJob("mybatchjob");
CloudTask myMultiInstanceTask = boundJob.GetTask("mymultiinstancetask");

// Now obtain the list of subtasks for the task
IPagedEnumerable<SubtaskInformation> subtasks = myMultiInstanceTask.ListSubtasks();

// Asynchronously iterate over the subtasks and print their stdout and stderr
// output if the subtask has completed
await subtasks.ForEachAsync(async (subtask) =>
{
    Console.WriteLine("subtask: {0}", subtask.Id);
    Console.WriteLine("exit code: {0}", subtask.ExitCode);

    if (subtask.State == SubtaskState.Completed)
    {
        ComputeNode node =
            await batchClient.PoolOperations.GetComputeNodeAsync(subtask.ComputeNodeInformation.PoolId,
subtask.ComputeNodeInformation.ComputeNodeId);

        NodeFile stdOutFile = await node.GetNodeFileAsync(subtask.ComputeNodeInformation.TaskRootDirectory +
"\\" + Constants.StandardOutFileName);
        NodeFile stdErrFile = await node.GetNodeFileAsync(subtask.ComputeNodeInformation.TaskRootDirectory +
"\\" + Constants.StandardErrorFileName);
        stdOut = await stdOutFile.ReadAsStringAsync();
        stdErr = await stdErrFile.ReadAsStringAsync();

        Console.WriteLine("node: {0}:", node.Id);
        Console.WriteLine("stdout.txt: {0}", stdOut);
        Console.WriteLine("stderr.txt: {0}", stdErr);
    }
    else
    {
        Console.WriteLine("\tSubtask {0} is in state {1}", subtask.Id, subtask.State);
    }
});
```

Code sample

The [MultiInstanceTasks](#) code sample on GitHub demonstrates how to use a multi-instance task to run an [MS-MPI](#) application on Batch compute nodes. Follow the steps in [Preparation](#) and [Execution](#) to run the sample.

Preparation

1. Follow the first two steps in [How to compile and run a simple MS-MPI program](#). This satisfies the prerequisites for the following step.
2. Build a *Release* version of the [MPIHelloWorld](#) sample MPI program. This is the program that will be run on

compute nodes by the multi-instance task.

3. Create a zip file containing `MPIHelloWorld.exe` (which you built step 2) and `MSMpiSetup.exe` (which you downloaded step 1). You'll upload this zip file as an application package in the next step.
4. Use the [Azure portal](#) to create a Batch [application](#) called "MPIHelloWorld", and specify the zip file you created in the previous step as version "1.0" of the application package. See [Upload and manage applications](#) for more information.

TIP

Build a *Release* version of `MPIHelloWorld.exe` so that you don't have to include any additional dependencies (for example, `msvcp140d.dll` or `vcruntime140d.dll`) in your application package.

Execution

1. Download the [azure-batch-samples](#) from GitHub.
2. Open the `MultInstanceTasks` **solution** in Visual Studio 2015 or newer. The `MultInstanceTasks.sln` solution file is located in:

```
azure-batch-samples\CSsharp\ArticleProjects\MultInstanceTasks\
```
3. Enter your Batch and Storage account credentials in `AccountSettings.settings` in the **Microsoft.Azure.Batch.Samples.Common** project.
4. **Build and run** the `MultInstanceTasks` solution to execute the MPI sample application on compute nodes in a Batch pool.
5. *Optional:* Use the [Azure portal](#) or the [Batch Explorer](#) to examine the sample pool, job, and task ("MultiInstanceSamplePool", "MultiInstanceSampleJob", "MultiInstanceSampleTask") before you delete the resources.

TIP

You can download [Visual Studio Community](#) for free if you do not have Visual Studio.

Output from `MultInstanceTasks.exe` is similar to the following:

```
Creating pool [MultiInstanceSamplePool]...
Creating job [MultiInstanceSampleJob]...
Adding task [MultiInstanceSampleTask] to job [MultiInstanceSampleJob]...
Awaiting task completion, timeout in 00:30:00...

Main task [MultiInstanceSampleTask] is in state [Completed] and ran on compute node [tvm-1219235766_1-20161017t162002z]:
---- stdout.txt ----
Rank 2 received string "Hello world" from Rank 0
Rank 1 received string "Hello world" from Rank 0

---- stderr.txt ----

Main task completed, waiting 00:00:10 for subtasks to complete...

---- Subtask information ----
subtask: 1
    exit code: 0
    node: tvm-1219235766_3-20161017t162002z
    stdout.txt:
    stderr.txt:
subtask: 2
    exit code: 0
    node: tvm-1219235766_2-20161017t162002z
    stdout.txt:
    stderr.txt:

Delete job? [yes] no: yes
Delete pool? [yes] no: yes

Sample complete, hit ENTER to exit...
```

Next steps

- The Microsoft HPC & Azure Batch Team blog discusses [MPI support for Linux on Azure Batch](#), and includes information on using [OpenFOAM](#) with Batch. You can find Python code samples for the [OpenFOAM example on GitHub](#).
- Learn how to [create pools of Linux compute nodes](#) for use in your Azure Batch MPI solutions.

Create task dependencies to run tasks that depend on other tasks

3/7/2017 • 8 min to read • [Edit Online](#)

You can define task dependencies to run a task or set of tasks only after a parent task has completed. Some scenarios where task dependencies are useful include:

- MapReduce-style workloads in the cloud.
- Jobs whose data processing tasks can be expressed as a directed acyclic graph (DAG).
- Pre-rendering and post-rendering processes, where each task must complete before the next task can begin.
- Any other job in which downstream tasks depend on the output of upstream tasks.

With Batch task dependencies, you can create tasks that are scheduled for execution on compute nodes after the completion of one or more parent tasks. For example, you can create a job that renders each frame of a 3D movie with separate, parallel tasks. The final task--the "merge task"--merges the rendered frames into the complete movie only after all frames have been successfully rendered.

By default, dependent tasks are scheduled for execution only after the parent task has completed successfully. You can specify a dependency action to override the default behavior and run tasks when the parent task fails. See the [Dependency actions](#) section for details.

You can create tasks that depend on other tasks in a one-to-one or one-to-many relationship. You can also create a range dependency where a task depends on the completion of a group of tasks within a specified range of task IDs. You can combine these three basic scenarios to create many-to-many relationships.

Task dependencies with Batch .NET

In this article, we discuss how to configure task dependencies by using the [Batch .NET](#) library. We first show you how to [enable task dependency](#) on your jobs, and then demonstrate how to [configure a task with dependencies](#). We also describe how to specify a dependency action to run dependent tasks if the parent fails. Finally, we discuss the [dependency scenarios](#) that Batch supports.

Enable task dependencies

To use task dependencies in your Batch application, you must first configure the job to use task dependencies. In Batch .NET, enable it on your [CloudJob](#) by setting its [UsesTaskDependencies](#) property to `true`:

```
CloudJob unboundJob = batchClient.JobOperations.CreateJob( "job001",
    new PoolInformation { PoolId = "pool001" });

// IMPORTANT: This is REQUIRED for using task dependencies.
unboundJob.UsesTaskDependencies = true;
```

In the preceding code snippet, "batchClient" is an instance of the [BatchClient](#) class.

Create dependent tasks

To create a task that depends on the completion of one or more parent tasks, you can specify that the task "depends on" the other tasks. In Batch .NET, configure the [CloudTask.DependsOn](#) property with an instance of the [TaskDependencies](#) class:

```
// Task 'Flowers' depends on completion of both 'Rain' and 'Sun'
// before it is run.
new CloudTask("Flowers", "cmd.exe /c echo Flowers")
{
    DependsOn = TaskDependencies.OnIds("Rain", "Sun")
},
```


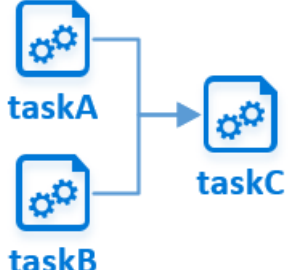

This code snippet creates a dependent task with task ID "Flowers". The "Flowers" task depends on tasks "Rain" and "Sun". Task "Flowers" will be scheduled to run on a compute node only after tasks "Rain" and "Sun" have completed successfully.

NOTE

A task is considered to be completed successfully when it is in the **completed** state and its **exit code** is `0`. In Batch .NET, this means a `CloudTask.State` property value of `Completed` and the `CloudTask's TaskExecutionInformation.ExitCode` property value is `0`.

Dependency scenarios

There are three basic task dependency scenarios that you can use in Azure Batch: one-to-one, one-to-many, and task ID range dependency. These can be combined to provide a fourth scenario, many-to-many.

SCENARIO	EXAMPLE	
One-to-one	<p><i>taskB</i> depends on <i>taskA</i></p> <p><i>taskB</i> will not be scheduled for execution until <i>taskA</i> has completed successfully</p>	 <pre>graph LR taskA[taskA] --> taskB[taskB]</pre>
One-to-many	<p><i>taskC</i> depends on both <i>taskA</i> and <i>taskB</i></p> <p><i>taskC</i> will not be scheduled for execution until both <i>taskA</i> and <i>taskB</i> have completed successfully</p>	 <pre>graph LR taskA[taskA] --> taskC[taskC] taskB[taskB] --> taskC</pre>
Task ID range	<p><i>taskD</i> depends on a range of tasks</p> <p><i>taskD</i> will not be scheduled for execution until the tasks with IDs 1 through 10 have completed successfully</p>	 <pre>graph LR tasks[Tasks 1-10] --> taskD[taskD]</pre>

TIP

You can create **many-to-many** relationships, such as where tasks C, D, E, and F each depend on tasks A and B. This is useful, for example, in parallelized preprocessing scenarios where your downstream tasks depend on the output of multiple upstream tasks.

In the examples in this section, a dependent task runs only after the parent tasks complete successfully. This behavior is the default behavior for a dependent task. You can run a dependent task after a parent task fails by specifying a dependency action to override the default behavior. See the [Dependency actions](#) section for details.

One-to-one

In a one-to-one relationship, a task depends on the successful completion of one parent task. To create the dependency, provide a single task ID to the [TaskDependencies.OnId](#) static method when you populate the [DependsOn](#) property of [CloudTask](#).

```
// Task 'taskA' doesn't depend on any other tasks
new CloudTask("taskA", "cmd.exe /c echo taskA"),

// Task 'taskB' depends on completion of task 'taskA'
new CloudTask("taskB", "cmd.exe /c echo taskB")
{
    DependsOn = TaskDependencies.OnId("taskA")
},
```

One-to-many

In a one-to-many relationship, a task depends on the completion of multiple parent tasks. To create the dependency, provide a collection of task IDs to the [TaskDependencies.OnIds](#) static method when you populate the [DependsOn](#) property of [CloudTask](#).

```
// 'Rain' and 'Sun' don't depend on any other tasks
new CloudTask("Rain", "cmd.exe /c echo Rain"),
new CloudTask("Sun", "cmd.exe /c echo Sun"),

// Task 'Flowers' depends on completion of both 'Rain' and 'Sun'
// before it is run.
new CloudTask("Flowers", "cmd.exe /c echo Flowers")
{
    DependsOn = TaskDependencies.OnIds("Rain", "Sun")
},
```

Task ID range

In a dependency on a range of parent tasks, a task depends on the the completion of tasks whose IDs lie within a range. To create the dependency, provide the first and last task IDs in the range to the [TaskDependencies.OnIdRange](#) static method when you populate the [DependsOn](#) property of [CloudTask](#).

IMPORTANT

When you use task ID ranges for your dependencies, the task IDs in the range *must* be string representations of integer values.

Every task in the range must satisfy the dependency, either by completing successfully or by completing with a failure that's mapped to a dependency action set to **Satisfy**. See the [Dependency actions](#) section for details.

```
// Tasks 1, 2, and 3 don't depend on any other tasks. Because
// we will be using them for a task range dependency, we must
// specify string representations of integers as their ids.
new CloudTask("1", "cmd.exe /c echo 1"),
new CloudTask("2", "cmd.exe /c echo 2"),
new CloudTask("3", "cmd.exe /c echo 3"),

// Task 4 depends on a range of tasks, 1 through 3
new CloudTask("4", "cmd.exe /c echo 4")
{
    // To use a range of tasks, their ids must be integer values.
    // Note that we pass integers as parameters to TaskIdRange,
    // but their ids (above) are string representations of the ids.
    DependsOn = TaskDependencies.OnIdRange(1, 3)
},
```

Dependency actions

By default, a dependent task or set of tasks runs only after a parent task has completed successfully. In some scenarios, you may want to run dependent tasks even if the parent task fails. You can override the default behavior by specifying a dependency action. A dependency action specifies whether a dependent task is eligible to run, based on the success or failure of the parent task.

For example, suppose that a dependent task is awaiting data from the completion of the upstream task. If the upstream task fails, the dependent task may still be able to run using older data. In this case, a dependency action can specify that the dependent task is eligible to run despite the failure of the parent task.

A dependency action is based on an exit condition for the parent task. You can specify a dependency action for any of the following exit conditions; for .NET, see the [ExitConditions](#) class for details:

- When a scheduling error occurs
- When the task exits with an exit code defined by the **ExitCodes** property
- When the task exits with an exit code that falls within a range specified by the **ExitCodeRanges** property
- The default case, if the task exits with an exit code not defined by **ExitCodes** or **ExitCodeRanges**, or if the task exits with a scheduling error and the **SchedulingError** property is not set

To specify a dependency action in .NET, set the [ExitOptions.DependencyAction](#) property for the exit condition. The **DependencyAction** property takes one of two values:

- Setting the **DependencyAction** property to **Satisfy** indicates that dependent tasks are eligible to run if the parent task exits with a specified error.
- Setting the **DependencyAction** property to **Block** indicates that dependent tasks are not eligible to run.

The default setting for the **DependencyAction** property is **Satisfy** for exit code 0, and **Block** for all other exit conditions.

The following code snippet sets the **DependencyAction** property for a parent task. If the parent task exits with a scheduling error, or with the specified error codes, the dependent task is blocked. If the parent task exits with any other non-zero error, the dependent task is eligible to run.

```
// Task A is the parent task.
new CloudTask("A", "cmd.exe /c echo A")
{
    // Specify exit conditions for task A and their dependency actions.
    ExitConditions = new ExitConditions()
    {
        // If task A exits with a scheduling error, block any downstream tasks (in this example, task B).
        SchedulingError = new ExitOptions()
        {
            DependencyAction = DependencyAction.Block
        },
        // If task A exits with the specified error codes, block any downstream tasks (in this example, task
        B).
        ExitCodes = new List<ExitCodeMapping>()
        {
            new ExitCodeMapping(10, new ExitOptions() { DependencyAction = DependencyAction.Block }),
            new ExitCodeMapping(20, new ExitOptions() { DependencyAction = DependencyAction.Block })
        },
        // If task A succeeds or fails with any other error, any downstream tasks become eligible to run
        // (in this example, task B).
        Default = new ExitOptions()
        {
            DependencyAction = DependencyAction.Satisfy
        }
    }
},
// Task B depends on task A. Whether it becomes eligible to run depends on how task A exits.
new CloudTask("B", "cmd.exe /c echo B")
{
    DependsOn = TaskDependencies.OnId("A")
},
```

Code sample

The [TaskDependencies](#) sample project is one of the [Azure Batch code samples](#) on GitHub. This Visual Studio solution demonstrates:

- How to enable task dependency on a job
- How to create tasks that depend on other tasks
- How to execute those tasks on a pool of compute nodes.

Next steps

Application deployment

The [application packages](#) feature of Batch provides an easy way to both deploy and version the applications that your tasks execute on compute nodes.

Installing applications and staging data

See [Installing applications and staging data on Batch compute nodes](#) in the Azure Batch forum for an overview of methods for preparing your nodes to run tasks. Written by one of the Azure Batch team members, this post is a good primer on the different ways to copy applications, task input data, and other files to your compute nodes.

Use Visual Studio project templates to jump-start Batch solutions

3/8/2017 • 22 min to read • [Edit Online](#)

The **Job Manager** and **Task Processor Visual Studio templates** for Batch provide code to help you to implement and run your compute-intensive workloads on Batch with the least amount of effort. This document describes these templates and provides guidance for how to use them.

IMPORTANT

This article discusses only information applicable to these two templates, and assumes that you are familiar with the Batch service and key concepts related to it: pools, compute nodes, jobs and tasks, job manager tasks, environment variables, and other relevant information. You can find more information in [Basics of Azure Batch](#), [Batch feature overview for developers](#), and [Get started with the Azure Batch library for .NET](#).

High-level overview

The Job Manager and Task Processor templates can be used to create two useful components:

- A job manager task that implements a job splitter that can break a job down into multiple tasks that can run independently, in parallel.
- A task processor that can be used to perform pre-processing and post-processing around an application command line.

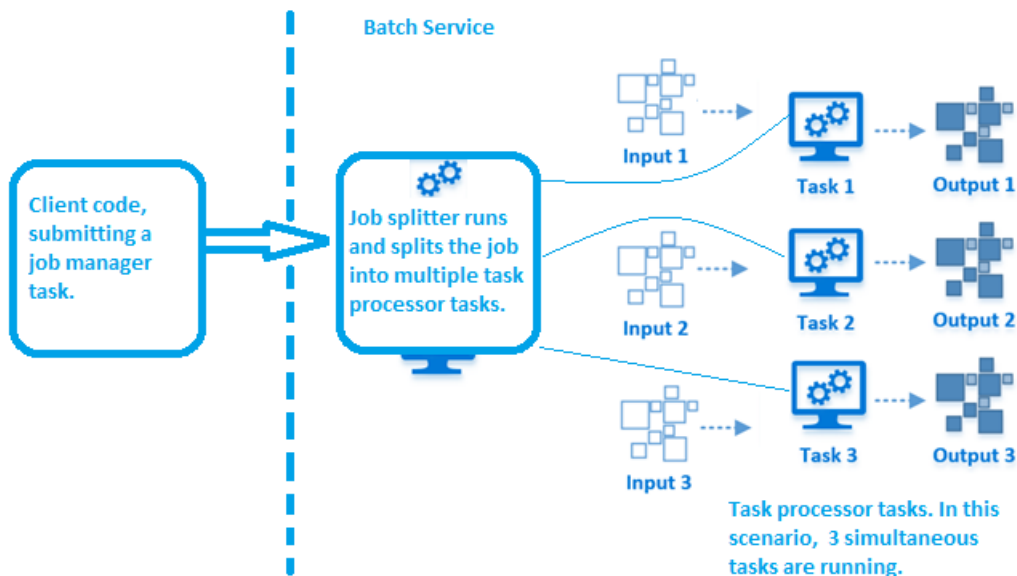
For example, in a movie rendering scenario, the job splitter would turn a single movie job into hundreds or thousands of separate tasks that would process individual frames separately. Correspondingly, the task processor would invoke the rendering application and all dependent processes that are needed to render each frame, as well as perform any additional actions (for example, copying the rendered frame to a storage location).

NOTE

The Job Manager and Task Processor templates are independent of each other, so you can choose to use both, or only one of them, depending on the requirements of your compute job and on your preferences.

As shown in the diagram below, a compute job that uses these templates will go through three stages:

1. The client code (e.g., application, web service, etc.) submits a job to the Batch service on Azure, specifying as its job manager task the job manager program.
2. The Batch service runs the job manager task on a compute node and the job splitter launches the specified number of task processor tasks, on as many compute nodes as required, based on the parameters and specifications in the job splitter code.
3. The task processor tasks run independently, in parallel, to process the input data and generate the output data.



Prerequisites

To use the Batch templates, you will need the following:

- A computer with Visual Studio 2015 or newer installed.
- The Batch templates, which are available from the [Visual Studio Gallery](#) as Visual Studio extensions. There are two ways to get the templates:
 - Install the templates using the **Extensions and Updates** dialog box in Visual Studio (for more information, see [Finding and Using Visual Studio Extensions](#)). In the **Extensions and Updates** dialog box, search and download the following two extensions:
 - Azure Batch Job Manager with Job Splitter
 - Azure Batch Task Processor
 - Download the templates from the online gallery for Visual Studio: [Microsoft Azure Batch Project Templates](#)
- If you plan to use the [Application Packages](#) feature to deploy the job manager and task processor to the Batch compute nodes, you need to link a storage account to your Batch account.

Preparation

We recommend creating a solution that can contain your job manager as well as your task processor, because this can make it easier to share code between your job manager and task processor programs. To create this solution, follow these steps:

1. Open Visual Studio and select **File > New > Project**.
2. Under **Templates**, expand **Other Project Types**, click **Visual Studio Solutions**, and then select **Blank Solution**.
3. Type a name that describes your application and the purpose of this solution (e.g., "LitwareBatchTaskPrograms").
4. To create the new solution, click **OK**.

Job Manager template

The Job Manager template helps you to implement a job manager task that can perform the following actions:

- Split a job into multiple tasks.
- Submit those tasks to run on Batch.

NOTE

For more information about job manager tasks, see [Batch feature overview for developers](#).

Create a Job Manager using the template

To add a job manager to the solution that you created earlier, follow these steps:

1. Open your existing solution in Visual Studio.
2. In Solution Explorer, right-click the solution, click **Add > New Project**.
3. Under **Visual C#**, click **Cloud**, and then click **Azure Batch Job Manager with Job Splitter**.
4. Type a name that describes your application and identifies this project as the job manager (e.g. "LitwareJobManager").
5. To create the project, click **OK**.
6. Finally, build the project to force Visual Studio to load all referenced NuGet packages and to verify that the project is valid before you start modifying it.

Job Manager template files and their purpose

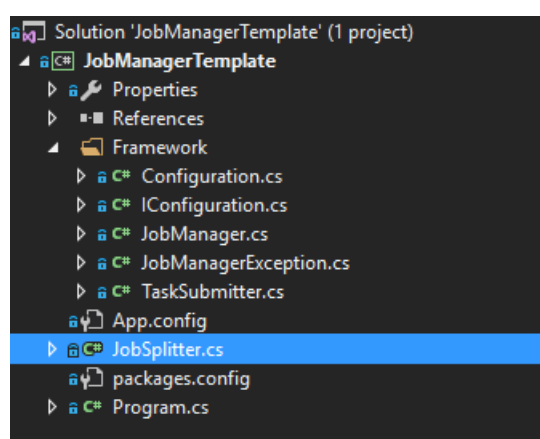
When you create a project using the Job Manager template, it generates three groups of code files:

- The main program file (Program.cs). This contains the program entry point and top-level exception handling. You shouldn't normally need to modify this.
- The Framework directory. This contains the files responsible for the 'boilerplate' work done by the job manager program – unpacking parameters, adding tasks to the Batch job, etc. You shouldn't normally need to modify these files.
- The job splitter file (JobSplitter.cs). This is where you will put your application-specific logic for splitting a job into tasks.

Of course you can add additional files as required to support your job splitter code, depending on the complexity of the job splitting logic.

The template also generates standard .NET project files such as a .csproj file, app.config, packages.config, etc.

The rest of this section describes the different files and their code structure, and explains what each class does.



Framework files

- `Configuration.cs`: Encapsulates the loading of job configuration data such as Batch account details, linked storage account credentials, job and task information, and job parameters. It also provides access to Batch-defined environment variables (see Environment settings for tasks, in the Batch documentation) via the `Configuration.EnvironmentVariable` class.
- `IConfiguration.cs`: Abstracts the implementation of the Configuration class, so that you can unit test your job splitter using a fake or mock configuration object.

- `JobManager.cs` : Orchestrates the components of the job manager program. It is responsible for the initializing the job splitter, invoking the job splitter, and dispatching the tasks returned by the job splitter to the task submitter.
- `JobManagerException.cs` : Represents an error that requires the job manager to terminate. `JobManagerException` is used to wrap 'expected' errors where specific diagnostic information can be provided as part of termination.
- `TaskSubmitter.cs` : This class is responsible to adding tasks returned by the job splitter to the Batch job. The `JobManager` class aggregates the sequence of tasks into batches for efficient but timely addition to the job, then calls `TaskSubmitter.SubmitTasks` on a background thread for each batch.

Job Splitter

`JobSplitter.cs` : This class contains application-specific logic for splitting the job into tasks. The framework invokes the `JobSplitter.Split` method to obtain a sequence of tasks, which it adds to the job as the method returns them. This is the class where you will inject the logic of your job. Implement the `Split` method to return a sequence of `CloudTask` instances representing the tasks into which you want to partition the job.

Standard .NET command line project files

- `App.config` : Standard .NET application configuration file.
- `Packages.config` : Standard NuGet package dependency file.
- `Program.cs` : Contains the program entry point and top-level exception handling.

Implementing the job splitter

When you open the Job Manager template project, the project will have the `JobSplitter.cs` file open by default. You can implement the split logic for the tasks in your workload by using the `Split()` method shown below:

```
/// <summary>
/// Gets the tasks into which to split the job. This is where you inject
/// your application-specific logic for decomposing the job into tasks.
///
/// The job manager framework invokes the Split method for you; you need
/// only to implement it, not to call it yourself. Typically, your
/// implementation should return tasks lazily, for example using a C#
/// iterator and the "yield return" statement; this allows tasks to be added
/// and to start running while splitting is still in progress.
/// </summary>
/// <returns>The tasks to be added to the job. Tasks are added automatically
/// by the job manager framework as they are returned by this method.</returns>
public IEnumerable<CloudTask> Split()
{
    // Your code for the split logic goes here.
    int startFrame = Convert.ToInt32(_parameters["StartFrame"]);
    int endFrame = Convert.ToInt32(_parameters["EndFrame"]);

    for (int i = startFrame; i <= endFrame; i++)
    {
        yield return new CloudTask("myTask" + i, "cmd /c dir");
    }
}
```

NOTE

The annotated section in the `Split()` method is the only section of the Job Manager template code that is intended for you to modify by adding the logic to split your jobs into different tasks. If you want to modify a different section of the template, please ensure you are familiarized with how Batch works, and try out a few of the [Batch code samples](#).

Your `Split()` implementation has access to:

- The job parameters, via the `_parameters` field.
- The CloudJob object representing the job, via the `_job` field.
- The CloudTask object representing the job manager task, via the `_jobManagerTask` field.

Your `Split()` implementation does not need to add tasks to the job directly. Instead, your code should return a sequence of CloudTask objects, and these will be added to the job automatically by the framework classes that invoke the job splitter. It's common to use C#'s iterator (`yield return`) feature to implement job splitters as this allows the tasks to start running as soon as possible rather than waiting for all tasks to be calculated.

Job splitter failure

If your job splitter encounters an error, it should either:

- Terminate the sequence using the C# `yield break` statement, in which case the job manager will be treated as successful; or
- Throw an exception, in which case the job manager will be treated as failed and may be retried depending on how the client has configured it).

In both cases, any tasks already returned by the job splitter and added to the Batch job will be eligible to run. If you don't want this to happen, then you could:

- Terminate the job before returning from the job splitter
- Formulate the entire task collection before returning it (that is, return an `ICollection<CloudTask>` or `IList<CloudTask>` instead of implementing your job splitter using a C# iterator)
- Use task dependencies to make all tasks depend on the successful completion of the job manager

Job manager retries

If the job manager fails, it may be retried by the Batch service depending on the client retry settings. In general, this is safe, because when the framework adds tasks to the job, it ignores any tasks that already exist. However, if calculating tasks is expensive, you may not wish to incur the cost of recalculating tasks that have already been added to the job; conversely, if the re-run is not guaranteed to generate the same task IDs then the 'ignore duplicates' behavior will not kick in. In these cases you should design your job splitter to detect the work that has already been done and not repeat it, for example by performing a `CloudJob.ListTasks` before starting to yield tasks.

Exit codes and exceptions in the Job Manager template

Exit codes and exceptions provide a mechanism to determine the outcome of running a program, and they can help to identify any problems with the execution of the program. The Job Manager template implements the exit codes and exceptions described in this section.

A job manager task that is implemented with the Job Manager template can return three possible exit codes:

CODE	DESCRIPTION
0	The job manager completed successfully. Your job splitter code ran to completion, and all tasks were added to the job.
1	The job manager task failed with an exception in an 'expected' part of the program. The exception was translated to a <code>JobManagerException</code> with diagnostic information and, where possible, suggestions for resolving the failure.
2	The job manager task failed with an 'unexpected' exception. The exception was logged to standard output, but the job manager was unable to add any additional diagnostic or remediation information.

In the case of job manager task failure, some tasks may still have been added to the service before the error occurred. These tasks will run as normal. See "Job Splitter Failure" above for discussion of this code path.

All the information returned by exceptions is written into stdout.txt and stderr.txt files. For more information, see [Error Handling](#).

Client considerations

This section describes some client implementation requirements when invoking a job manager based on this template. See [How to pass parameters and environment variables from the client code](#) for details on passing parameters and environment settings.

Mandatory credentials

In order to add tasks to the Azure Batch job, the job manager task requires your Azure Batch account URL and key. You must pass these in environment variables named YOUR_BATCH_URL and YOUR_BATCH_KEY. You can set these in the Job Manager task environment settings. For example, in a C# client:

```
job.JobManagerTask.EnvironmentSettings = new [] {  
    new EnvironmentSetting("YOUR_BATCH_URL", "https://account.region.batch.azure.com"),  
    new EnvironmentSetting("YOUR_BATCH_KEY", "{your_base64_encoded_account_key}"),  
};
```

Storage credentials

Typically, the client does not need to provide the linked storage account credentials to the job manager task because (a) most job managers do not need to explicitly access the linked storage account and (b) the linked storage account is often provided to all tasks as a common environment setting for the job. If you are not providing the linked storage account via the common environment settings, and the job manager requires access to linked storage, then you should supply the linked storage credentials as follows:

```
job.JobManagerTask.EnvironmentSettings = new [] {  
    /* other environment settings */  
    new EnvironmentSetting("LINKED_STORAGE_ACCOUNT", "{storageAccountName}"),  
    new EnvironmentSetting("LINKED_STORAGE_KEY", "{storageAccountKey}"),  
};
```

Job manager task settings

The client should set the job manager *killJobOnCompletion* flag to **false**.

It is usually safe for the client to set *runExclusive* to **false**.

The client should use the *resourceFiles* or *applicationPackageReferences* collection to have the job manager executable (and its required DLLs) deployed to the compute node.

By default, the job manager will not be retried if it fails. Depending on your job manager logic, the client may want to enable retries via *constraints/maxTaskRetryCount*.

Job settings

If the job splitter emits tasks with dependencies, the client must set the job's *usesTaskDependencies* to true.

In the job splitter model, it is unusual for clients to wish to add tasks to jobs over and above what the job splitter creates. The client should therefore normally set the job's *onAllTasksComplete* to **terminatejob**.

Task Processor template

A Task Processor template helps you to implement a task processor that can perform the following actions:

- Set up the information required by each Batch task to run.
- Run all actions required by each Batch task.
- Save task outputs to persistent storage.

Although a task processor is not required to run tasks on Batch, the key advantage of using a task processor is that it provides a wrapper to implement all task execution actions in one location. For example, if you need to run several applications in the context of each task, or if you need to copy data to persistent storage after completing each task.

The actions performed by the task processor can be as simple or complex, and as many or as few, as required by your workload. Additionally, by implementing all task actions into one task processor, you can readily update or add actions based on changes to applications or workload requirements. However, in some cases a task processor might not be the optimal solution for your implementation as it can add unnecessary complexity, for example when running jobs that can be quickly started from a simple command line.

Create a Task Processor using the template

To add a task processor to the solution that you created earlier, follow these steps:

1. Open your existing solution in Visual Studio.
2. In Solution Explorer, right-click the solution, click **Add**, and then click **New Project**.
3. Under **Visual C#**, click **Cloud**, and then click **Azure Batch Task Processor**.
4. Type a name that describes your application and identifies this project as the task processor (e.g. "LitwareTaskProcessor").
5. To create the project, click **OK**.
6. Finally, build the project to force Visual Studio to load all referenced NuGet packages and to verify that the project is valid before you start modifying it.

Task Processor template files and their purpose

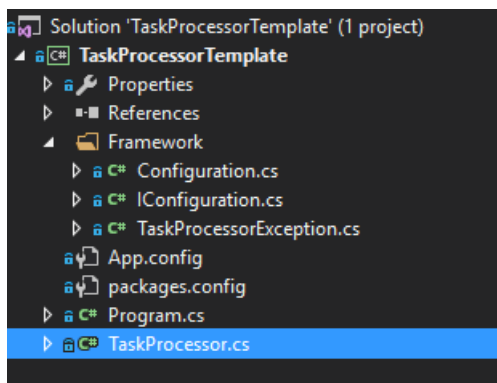
When you create a project using the task processor template, it generates three groups of code files:

- The main program file (Program.cs). This contains the program entry point and top-level exception handling. You shouldn't normally need to modify this.
- The Framework directory. This contains the files responsible for the 'boilerplate' work done by the job manager program – unpacking parameters, adding tasks to the Batch job, etc. You shouldn't normally need to modify these files.
- The task processor file (TaskProcessor.cs). This is where you will put your application-specific logic for executing a task (typically by calling out to an existing executable). Pre- and post-processing code, such as downloading additional data or uploading result files, also goes here.

Of course you can add additional files as required to support your task processor code, depending on the complexity of the job splitting logic.

The template also generates standard .NET project files such as a .csproj file, app.config, packages.config, etc.

The rest of this section describes the different files and their code structure, and explains what each class does.



Framework files

- `Configuration.cs`: Encapsulates the loading of job configuration data such as Batch account details, linked storage account credentials, job and task information, and job parameters. It also provides access to Batch-defined environment variables (see Environment settings for tasks, in the Batch documentation) via the `Configuration.EnvironmentVariable` class.
- `IConfiguration.cs`: Abstracts the implementation of the `Configuration` class, so that you can unit test your job splitter using a fake or mock configuration object.
- `TaskProcessorException.cs`: Represents an error that requires the job manager to terminate. `TaskProcessorException` is used to wrap 'expected' errors where specific diagnostic information can be provided as part of termination.

Task Processor

- `TaskProcessor.cs`: Runs the task. The framework invokes the `TaskProcessor.Run` method. This is the class where you will inject the application-specific logic of your task. Implement the `Run` method to:
 - Parse and validate any task parameters
 - Compose the command line for any external program you want to invoke
 - Log any diagnostic information you may require for debugging purposes
 - Start a process using that command line
 - Wait for the process to exit
 - Capture the exit code of the process to determine if it succeeded or failed
 - Save any output files you want to keep to persistent storage

Standard .NET command line project files

- `App.config`: Standard .NET application configuration file.
- `Packages.config`: Standard NuGet package dependency file.
- `Program.cs`: Contains the program entry point and top-level exception handling.

Implementing the task processor

When you open the Task Processor template project, the project will have the `TaskProcessor.cs` file open by default. You can implement the run logic for the tasks in your workload by using the `Run()` method shown below:

```

/// <summary>
/// Runs the task processing logic. This is where you inject
/// your application-specific logic for decomposing the job into tasks.
///
/// The task processor framework invokes the Run method for you; you need
/// only to implement it, not to call it yourself. Typically, your
/// implementation will execute an external program (from resource files or
/// an application package), check the exit code of that program and
/// save output files to persistent storage.
/// </summary>
public async Task<int> Run()

{
    try
    {
        //Your code for the task processor goes here.
        var command = $"compare {_parameters["Frame1"]} {_parameters["Frame2"]} compare.gif";
        using (var process = Process.Start($"cmd /c {command}"))
        {
            process.WaitForExit();
            var taskOutputStorage = new TaskOutputStorage(
                _configuration.StorageAccount,
                _configuration.JobId,
                _configuration.TaskId
            );
            await taskOutputStorage.SaveAsync(
                TaskOutputKind.TaskOutput,
                @".\stdout.txt",
                @"stdout.txt"
            );
            return process.ExitCode;
        }
    }
    catch (Exception ex)
    {
        throw new TaskProcessorException(
            $"{ex.GetType().Name} exception in run task processor: {ex.Message}",
            ex
        );
    }
}

```

NOTE

The annotated section in the `Run()` method is the only section of the Task Processor template code that is intended for you to modify by adding the run logic for the tasks in your workload. If you want to modify a different section of the template, please first familiarize yourself with how Batch works by reviewing the Batch documentation and trying out a few of the Batch code samples.

The `Run()` method is responsible for launching the command line, starting one or more processes, waiting for all process to complete, saving the results, and finally returning with an exit code. The `Run()` method is where you implement the processing logic for your tasks. The task processor framework invokes the `Run()` method for you; you do not need to call it yourself.

Your `Run()` implementation has access to:

- The task parameters, via the `_parameters` field.
- The job and task ids, via the `_jobId` and `_taskId` fields.
- The task configuration, via the `_configuration` field.

Task failure

In case of failure, you can exit the Run() method by throwing an exception, but this leaves the top level exception handler in control of the task exit code. If you need to control the exit code so that you can distinguish different types of failure, for example for diagnostic purposes or because some failure modes should terminate the job and others should not, then you should exit the Run() method by returning a non-zero exit code. This becomes the task exit code.

Exit codes and exceptions in the Task Processor template

Exit codes and exceptions provide a mechanism to determine the outcome of running a program, and they can help identify any problems with the execution of the program. The Task Processor template implements the exit codes and exceptions described in this section.

A task processor task that is implemented with the Task Processor template can return three possible exit codes:

CODE	DESCRIPTION
Process.ExitCode	The task processor ran to completion. Note that this does not imply that the program you invoked was successful – only that the task processor invoked it successfully and performed any post-processing without exceptions. The meaning of the exit code depends on the invoked program – typically exit code 0 means the program succeeded and any other exit code means the program failed.
1	The task processor failed with an exception in an 'expected' part of the program. The exception was translated to a <code>TaskProcessorException</code> with diagnostic information and, where possible, suggestions for resolving the failure.
2	The task processor failed with an 'unexpected' exception. The exception was logged to standard output, but the task processor was unable to add any additional diagnostic or remediation information.

NOTE

If the program you invoke uses exit codes 1 and 2 to indicate specific failure modes, then using exit codes 1 and 2 for task processor errors is ambiguous. You can change these task processor error codes to distinctive exit codes by editing the exception cases in the Program.cs file.

All the information returned by exceptions is written into stdout.txt and stderr.txt files. For more information, see Error Handling, in the Batch documentation.

Client considerations

Storage credentials

If your task processor uses Azure blob storage to persist outputs, for example using the file conventions helper library, then it needs access to *either* the cloud storage account credentials *or* a blob container URL that includes a shared access signature (SAS). The template includes support for providing credentials via common environment variables. Your client can pass the storage credentials as follows:

```
job.CommonEnvironmentSettings = new [] {  
    new EnvironmentSetting("LINKED_STORAGE_ACCOUNT", "{storageAccountName}"),  
    new EnvironmentSetting("LINKED_STORAGE_KEY", "{storageAccountKey}"),  
};
```

The storage account is then available in the TaskProcessor class via the `_configuration.StorageAccount` property.

If you prefer to use a container URL with SAS, you can also pass this via an job common environment setting, but the task processor template does not currently include built-in support for this.

Storage setup

It is recommended that the client or job manager task create any containers required by tasks before adding the tasks to the job. This is mandatory if you use a container URL with SAS, as such a URL does not include permission to create the container. It is recommended even if you pass storage account credentials, as it saves every task having to call `CloudBlobContainer.CreateIfNotExistsAsync` on the container.

Pass parameters and environment variables

Pass environment settings

A client can pass information to the job manager task in the form of environment settings. This information can then be used by the job manager task when generating the task processor tasks that will run as part of the compute job. Examples of the information that you can pass as environment settings are:

- Storage account name and account keys
- Batch account URL
- Batch account key

The Batch service has a simple mechanism to pass environment settings to a job manager task by using the `EnvironmentSettings` property in [Microsoft.Azure.Batch.JobManagerTask](#).

For example, to get the `BatchClient` instance for a Batch account, you can pass as environment variables from the client code the URL and shared key credentials for the Batch account. Likewise, to access the storage account that is linked to the Batch account, you can pass the storage account name and the storage account key as environment variables.

Pass parameters to the Job Manager template

In many cases, it's useful to pass per-job parameters to the job manager task, either to control the job splitting process or to configure the tasks for the job. You can do this by uploading a JSON file named `parameters.json` as a resource file for the job manager task. The parameters can then become available in the `JobSplitter._parameters` field in the Job Manager template.

NOTE

The built-in parameter handler supports only string-to-string dictionaries. If you want to pass complex JSON values as parameter values, you will need to pass these as strings and parse them in the job splitter, or modify the framework's

`Configuration.GetJobParameters` method.

Pass parameters to the Task Processor template

You can also pass parameters to individual tasks implemented using the Task Processor template. Just as with the job manager template, the task processor template looks for a resource file named

`parameters.json`, and if found it loads it as the parameters dictionary. There are a couple of options for how to pass parameters to the task processor tasks:

- Reuse the job parameters JSON. This works well if the only parameters are job-wide ones (for example, a render height and width). To do this, when creating a `CloudTask` in the job splitter, add a reference to the `parameters.json` resource file object from the job manager task's `ResourceFiles` (`JobSplitter._jobManagerTask.ResourceFiles`) to the `CloudTask`'s `ResourceFiles` collection.
- Generate and upload a task-specific `parameters.json` document as part of job splitter execution, and reference that blob in the task's resource files collection. This is necessary if different tasks have different parameters. An

example might be a 3D rendering scenario where the frame index is passed to the task as a parameter.

NOTE

The built-in parameter handler supports only string-to-string dictionaries. If you want to pass complex JSON values as parameter values, you will need to pass these as strings and parse them in the task processor, or modify the framework's `Configuration.GetTaskParameters` method.

Next steps

Persist job and task output to Azure Storage

Another helpful tool in Batch solution development is [Azure Batch File Conventions](#). Use this .NET class library (currently in preview) in your Batch .NET applications to easily store and retrieve task outputs to and from Azure Storage. [Persist Azure Batch job and task output](#) contains a full discussion of the library and its usage.

Batch Forum

The [Azure Batch Forum](#) on MSDN is a great place to discuss Batch and ask questions about the service. Head on over for helpful "sticky" posts, and post your questions as they arise while you build your Batch solutions.

Manage Batch resources with PowerShell cmdlets

2/27/2017 • 8 min to read • [Edit Online](#)

With the Azure Batch PowerShell cmdlets, you can perform and script many of the same tasks you carry out with the Batch APIs, the Azure portal, and the Azure Command-Line Interface (CLI). This is a quick introduction to the cmdlets you can use to manage your Batch accounts and work with your Batch resources such as pools, jobs, and tasks.

For a complete list of Batch cmdlets and detailed cmdlet syntax, see the [Azure Batch cmdlet reference](#).

This article is based on cmdlets in Azure PowerShell version 3.0.0. We recommend that you update your Azure PowerShell frequently to take advantage of service updates and enhancements.

Prerequisites

Perform the following operations to use Azure PowerShell to manage your Batch resources.

- [Install and configure Azure PowerShell](#)
- Run the **Login-AzureRmAccount** cmdlet to connect to your subscription (the Azure Batch cmdlets ship in the Azure Resource Manager module):

```
Login-AzureRmAccount
```

- **Register with the Batch provider namespace.** This operation only needs to be performed **once per subscription**.

```
Register-AzureRMResourceProvider -ProviderNamespace Microsoft.Batch
```

Manage Batch accounts and keys

Create a Batch account

New-AzureRmBatchAccount creates a Batch account in a specified resource group. If you don't already have a resource group, create one by running the [New-AzureRmResourceGroup](#) cmdlet. Specify one of the Azure regions in the **Location** parameter, such as "Central US". For example:

```
New-AzureRmResourceGroup -Name MyBatchResourceGroup -location "Central US"
```

Then, create a Batch account in the resource group, specifying a name for the account in *<account_name>* and the location and name of your resource group. Creating the Batch account can take some time to complete. For example:

```
New-AzureRmBatchAccount -AccountName <account_name> -Location "Central US" -ResourceGroupName <res_group_name>
```

NOTE

The Batch account name must be unique to the Azure region for the resource group, contain between 3 and 24 characters, and use lowercase letters and numbers only.

Get account access keys

Get-AzureRmBatchAccountKeys shows the access keys associated with an Azure Batch account. For example, run the following to get the primary and secondary keys of the account you created.

```
$Account = Get-AzureRmBatchAccountKeys -AccountName <account_name>

$Account.PrimaryAccountKey

$Account.SecondaryAccountKey
```

Generate a new access key

New-AzureRmBatchAccountKey generates a new primary or secondary account key for an Azure Batch account. For example, to generate a new primary key for your Batch account, type:

```
New-AzureRmBatchAccountKey -AccountName <account_name> -KeyType Primary
```

NOTE

To generate a new secondary key, specify "Secondary" for the **KeyType** parameter. You have to regenerate the primary and secondary keys separately.

Delete a Batch account

Remove-AzureRmBatchAccount deletes a Batch account. For example:

```
Remove-AzureRmBatchAccount -AccountName <account_name>
```

When prompted, confirm you want to remove the account. Account removal can take some time to complete.

Create a BatchAccountContext object

To authenticate using the Batch PowerShell cmdlets when you create and manage Batch pools, jobs, tasks, and other resources, first create a BatchAccountContext object to store your account name and keys:

```
$context = Get-AzureRmBatchAccountKeys -AccountName <account_name>
```

You pass the BatchAccountContext object into cmdlets that use the **BatchContext** parameter.

NOTE

By default, the account's primary key is used for authentication, but you can explicitly select the key to use by changing your BatchAccountContext object's **KeyInUse** property: `$context.KeyInUse = "Secondary"`.

Create and modify Batch resources

Use cmdlets such as **New-AzureBatchPool**, **New-AzureBatchJob**, and **New-AzureBatchTask** to create resources under a Batch account. There are corresponding **Get-** and **Set-** cmdlets to update the properties of existing resources, and **Remove-** cmdlets to remove resources under a Batch account.

When using many of these cmdlets, in addition to passing a BatchContext object, you need to create or pass objects that contain detailed resource settings, as shown in the following example. See the detailed help for each cmdlet for additional examples.

Create a Batch pool

When creating or updating a Batch pool, you select a cloud service configuration or a virtual machine configuration for the operating system on the compute nodes (see [Batch feature overview](#)). Your choice determines whether your compute nodes are imaged with one of the [Azure Guest OS releases](#) or with one of the supported Linux or Windows VM images in the Azure Marketplace.

When you run **New-AzureBatchPool**, pass the operating system settings in a `PSCloudServiceConfiguration` or `PSVirtualMachineConfiguration` object. For example, the following cmdlet creates a new Batch pool with size `Small` compute nodes in the cloud service configuration, imaged with the latest operating system version of family 3 (Windows Server 2012). Here, the **CloudServiceConfiguration** parameter specifies the *\$configuration* variable as the `PSCloudServiceConfiguration` object. The **BatchContext** parameter specifies a previously defined variable *\$context* as the `BatchAccountContext` object.

```
$configuration = New-Object -TypeName "Microsoft.Azure.Commands.Batch.Models.PSCloudServiceConfiguration" -ArgumentList @(4,"*")

New-AzureBatchPool -Id "AutoScalePool" -VirtualMachineSize "Small" -CloudServiceConfiguration $configuration -AutoScaleFormula '$TargetDedicated=4;' -BatchContext $context
```

The target number of compute nodes in the new pool is determined by an autoscaling formula. In this case, the formula is simply **\$TargetDedicated=4**, indicating the number of compute nodes in the pool is 4 at most.

Query for pools, jobs, tasks, and other details

Use cmdlets such as **Get-AzureBatchPool**, **Get-AzureBatchJob**, and **Get-AzureBatchTask** to query for entities created under a Batch account.

Query for data

As an example, use **Get-AzureBatchPools** to find your pools. By default this queries for all pools under your account, assuming you already stored the `BatchAccountContext` object in *\$context*:

```
Get-AzureBatchPool -BatchContext $context
```

Use an OData filter

You can supply an OData filter using the **Filter** parameter to find only the objects you're interested in. For example, you can find all pools with ids starting with "myPool":

```
$filter = "startswith(id,'myPool')"

Get-AzureBatchPool -Filter $filter -BatchContext $context
```

This method is not as flexible as using "Where-Object" in a local pipeline. However, the query gets sent to the Batch service directly so that all filtering happens on the server side, saving Internet bandwidth.

Use the Id parameter

An alternative to an OData filter is to use the **Id** parameter. To query for a specific pool with id "myPool":

```
Get-AzureBatchPool -Id "myPool" -BatchContext $context
```

The **Id** parameter supports only full-id search, not wildcards or OData-style filters.

Use the MaxCount parameter

By default, each cmdlet returns a maximum of 1000 objects. If you reach this limit, either refine your filter to bring

back fewer objects, or explicitly set a maximum using the **MaxCount** parameter. For example:

```
Get-AzureBatchTask -MaxCount 2500 -BatchContext $context
```

To remove the upper bound, set **MaxCount** to 0 or less.

Use the PowerShell pipeline

Batch cmdlets can leverage the PowerShell pipeline to send data between cmdlets. This has the same effect as specifying a parameter, but makes working with multiple entities easier.

For example, find and display all tasks under your account:

```
Get-AzureBatchJob -BatchContext $context | Get-AzureBatchTask -BatchContext $context
```

Restart (reboot) every compute node in a pool:

```
Get-AzureBatchComputeNode -PoolId "myPool" -BatchContext $context | Restart-AzureBatchComputeNode -  
BatchContext $context
```

Application package management

Application packages provide a simplified way to deploy applications to the compute nodes in your pools. With the Batch PowerShell cmdlets, you can upload and manage application packages in your Batch account, and deploy package versions to compute nodes.

Create an application:

```
New-AzureRmBatchApplication -AccountName <account_name> -ResourceGroupName <res_group_name> -ApplicationId  
"MyBatchApplication"
```

Add an application package:

```
New-AzureRmBatchApplicationPackage -AccountName <account_name> -ResourceGroupName <res_group_name> -  
ApplicationId "MyBatchApplication" -ApplicationVersion "1.0" -Format zip -FilePath package001.zip
```

Set the **default version** for the application:

```
Set-AzureRmBatchApplication -AccountName <account_name> -ResourceGroupName <res_group_name> -ApplicationId  
"MyBatchApplication" -DefaultVersion "1.0"
```

List an application's packages

```
$application = Get-AzureRmBatchApplication -AccountName <account_name> -ResourceGroupName <res_group_name> -  
ApplicationId "MyBatchApplication"  
  
$application.ApplicationPackages
```

Delete an application package

```
Remove-AzureRmBatchApplicationPackage -AccountName <account_name> -ResourceGroupName <res_group_name> -  
ApplicationId "MyBatchApplication" -ApplicationVersion "1.0"
```

Delete an application

```
Remove-AzureRmBatchApplication -AccountName <account_name> -ResourceGroupName <res_group_name> -ApplicationId "MyBatchApplication"
```

NOTE

You must delete all of an application's application package versions before you delete the application. You will receive a 'Conflict' error if you try to delete an application that currently has application packages.

Deploy an application package

You can specify one or more application packages for deployment when you create a pool. When you specify a package at pool creation time, it is deployed to each node as the node joins pool. Packages are also deployed when a node is rebooted or reimaged.

Specify the `-ApplicationPackageReference` option when creating a pool to deploy an application package to the pool's nodes as they join the pool. First, create a **PSApplicationPackageReference** object, and configure it with the application Id and package version you want to deploy to the pool's compute nodes:

```
$appPackageReference = New-Object Microsoft.Azure.Commands.Batch.Models.PSApplicationPackageReference  
  
$appPackageReference.ApplicationId = "MyBatchApplication"  
  
$appPackageReference.Version = "1.0"
```

Now create the pool, and specify the package reference object as the argument to the

`ApplicationPackageReferences` option:

```
New-AzureBatchPool -Id "PoolWithAppPackage" -VirtualMachineSize "Small" -CloudServiceConfiguration  
$configuration -BatchContext $context -ApplicationPackageReferences $appPackageReference
```

You can find more information on application packages in [Application deployment with Azure Batch application packages](#).

IMPORTANT

You must [link an Azure Storage account](#) to your Batch account to use application packages.

Update a pool's application packages

To update the applications assigned to an existing pool, first create a **PSApplicationPackageReference** object with the desired properties (application Id and package version):

```
$appPackageReference = New-Object Microsoft.Azure.Commands.Batch.Models.PSApplicationPackageReference  
  
$appPackageReference.ApplicationId = "MyBatchApplication"  
  
$appPackageReference.Version = "2.0"
```

Next, get the pool from Batch, clear out any existing packages, add our new package reference, and update the Batch service with the new pool settings:

```
$pool = Get-AzureBatchPool -BatchContext $context -Id "PoolWithAppPackage"

$pool.ApplicationPackageReferences.Clear()

$pool.ApplicationPackageReferences.Add($appPackageReference)

Set-AzureBatchPool -BatchContext $context -Pool $pool
```

You've now updated the pool's properties in the Batch service. To actually deploy the new application package to compute nodes in the pool, however, you must restart or reimagine those nodes. You can restart every node in a pool with this command:

```
Get-AzureBatchComputeNode -PoolId "PoolWithAppPackage" -BatchContext $context | Restart-AzureBatchComputeNode -BatchContext $context
```

TIP

You can deploy multiple application packages to the compute nodes in a pool. If you'd like to *add* an application package instead of replacing the currently deployed packages, omit the `$pool.ApplicationPackageReferences.Clear()` line above.

Next steps

- For detailed cmdlet syntax and examples, see [Azure Batch cmdlet reference](#).
- For more information about applications and application packages in Batch, see [Application deployment with Azure Batch application packages](#).

Manage Batch resources with Azure CLI

3/20/2017 • 10 min to read • [Edit Online](#)

The cross-platform Azure Command-Line Interface (Azure CLI) enables you to manage your Batch accounts and resources such as pools, jobs, and tasks in Linux, Mac, and Windows command shells. With the Azure CLI, you can perform and script many of the same tasks you carry out with the Batch APIs, Azure portal, and Batch PowerShell cmdlets.

This article is based on Azure CLI version 0.10.5.

Prerequisites

- [Install the Azure CLI](#)
- [Connect the Azure CLI to your Azure subscription](#)
- Switch to **Resource Manager mode**: `azure config mode arm`

TIP

We recommend that you update your Azure CLI installation frequently to take advantage of service updates and enhancements.

Command help

You can display help text for every command in the Azure CLI by appending `-h` as the only option after the command. For example:

- To get help for the `azure` command, enter: `azure -h`
- To get a list of all Batch commands in the CLI, use: `azure batch -h`
- To get help on creating a Batch account, enter: `azure batch account create -h`

When in doubt, use the `-h` command-line option to get help on any Azure CLI command.

Create a Batch account

Usage:

```
azure batch account create [options] <name>
```

Example:

```
azure batch account create --location "West US" --resource-group "resgroup001" "batchaccount001"
```

Creates a new Batch account with the specified parameters. You must specify at least a location, resource group, and account name. If you don't already have a resource group, create one by running `azure group create`, and specify one of the Azure regions (such as "West US") for the `--location` option. For example:

```
azure group create --name "resgroup001" --location "West US"
```


NOTE

The Batch account name must be unique within the Azure region the account is created. It may contain only lowercase alphanumeric characters, and must be 3-24 characters in length. You can't use special characters like `-` or `_` in Batch account names.

Linked storage account (autostorage)

You can (optionally) link a **General purpose** Storage account to your Batch account when you create it. The [application packages](#) feature of Batch uses blob storage in a linked General purpose Storage account, as does the [Batch File Conventions .NET](#) library. These optional features assist you in deploying the applications your Batch tasks run, and persisting the data they produce.

To link an existing Azure Storage account to a new Batch account when you create it, specify the

`--autostorage-account-id` option. This option requires the fully qualified resource ID of the storage account.

First, show your storage account's details:

```
azure storage account show --resource-group "resgroup001" "storageaccount001"
```

Then use the **Url** value for the `--autostorage-account-id` option. The Url value starts with `"/subscriptions/"` and contains your subscription ID and resource path to the Storage account:

```
azure batch account create --location "West US" --resource-group "resgroup001" --autostorage-account-id  
"/subscriptions/8fffffff8-4444-4444-bfbf-  
8fffffff84444/resourceGroups/resgroup001/providers/Microsoft.Storage/storageAccounts/storageaccount001"  
"batchaccount001"
```

Delete a Batch account

Usage:

```
azure batch account delete [options] <name>
```

Example:

```
azure batch account delete --resource-group "resgroup001" "batchaccount001"
```

Deletes the specified Batch account. When prompted, confirm you want to remove the account (account removal can take some time to complete).

Manage account access keys

You need an access key to [create and modify resources](#) in your Batch account.

List access keys

Usage:

```
azure batch account keys list [options] <name>
```

Example:

```
azure batch account keys list --resource-group "resgroup001" "batchaccount001"
```

Lists the account keys for the given Batch account.

Generate a new access key

Usage:

```
azure batch account keys renew [options] --<primary|secondary> <name>
```

Example:

```
azure batch account keys renew --resource-group "resgroup001" --primary "batchaccount001"
```

Regenerates the specified account key for the given Batch account.

Create and modify Batch resources

You can use the Azure CLI to create, read, update, and delete (CRUD) Batch resources like pools, compute nodes, jobs, and tasks. These CRUD operations require your Batch account name, access key, and endpoint. You can specify these with the `-a`, `-k`, and `-u` options, or set [environment variables](#) which the CLI uses automatically (if populated).

Credential environment variables

You can set `AZURE_BATCH_ACCOUNT`, `AZURE_BATCH_ACCESS_KEY`, and `AZURE_BATCH_ENDPOINT` environment variables instead of specifying `-a`, `-k`, and `-u` options on the command line for every command you execute. The Batch CLI uses these variables (if set) so that you can omit the `-a`, `-k`, and `-u` options. The remainder of this article assumes use of these environment variables.

TIP

List your keys with `azure batch account keys list`, and display the account's endpoint with `azure batch account show`.

JSON files

When you create Batch resources like pools and jobs, you can specify a JSON file containing the new resource's configuration instead of passing its parameters as command-line options. For example:

```
azure batch pool create my_batch_pool.json
```

While you can perform many resource creation operations using only command-line options, some features require a JSON-formatted file containing the resource details. For example, you must use a JSON file if you want to specify resource files for a start task.

To find the JSON required to create a resource, refer to the [Batch REST API reference](#) documentation on MSDN. Each "Add *resource type*" topic contains example JSON for creating the resource, which you can use as templates for your JSON files. For example, JSON for pool creation can be found in [Add a pool to an account](#).

NOTE

If you specify a JSON file when you create a resource, all other parameters that you specify on the command line for that resource are ignored.

Create a pool

Usage:

```
azure batch pool create [options] [json-file]
```

Example (Virtual Machine Configuration):

```
azure batch pool create --id "pool001" --target-dedicated 1 --vm-size "STANDARD_A1" --image-publisher  
"Canonical" --image-offer "UbuntuServer" --image-sku "14.04.2-LTS" --node-agent-id "batch.node.ubuntu 14.04"
```

Example (Cloud Services Configuration):

```
azure batch pool create --id "pool002" --target-dedicated 1 --vm-size "small" --os-family "4"
```

Creates a pool of compute nodes in the Batch service.

As mentioned in the [Batch feature overview](#), you have two options when you select an operating system for the nodes in your pool: **Virtual Machine Configuration** and **Cloud Services Configuration**. Use the `--image-*` options to create Virtual Machine Configuration pools, and `--os-family` to create Cloud Services Configuration pools. You can't specify both `--os-family` and `--image-*` options.

You can specify pool [application packages](#) and the command line for a [start task](#). To specify resource files for the start task, however, you must instead use a [JSON file](#).

Delete a pool with:

```
azure batch pool delete [pool-id]
```

TIP

Check the [list of virtual machine images](#) for values appropriate for the `--image-*` options.

Create a job

Usage:

```
azure batch job create [options] [json-file]
```

Example:

```
azure batch job create --id "job001" --pool-id "pool001"
```

Adds a job to the Batch account and specifies the pool on which its tasks execute.

Delete a job with:

```
azure batch job delete [job-id]
```

List pools, jobs, tasks, and other resources

Each Batch resource type supports a `list` command that queries your Batch account and lists resources of that type. For example, you can list the pools in your account and the tasks in a job:

```
azure batch pool list
azure batch task list --job-id "job001"
```

Listing resources efficiently

For faster querying, you can specify **select**, **filter**, and **expand** clause options for `list` operations. Use these options to limit the amount of data returned by the Batch service. Because all filtering occurs server-side, only the data you are interested in crosses the wire. Use these clauses to save bandwidth (and therefore time) when you perform list operations.

For example, this will return only pools whose ids start with "renderTask":

```
azure batch task list --job-id "job001" --filter-clause "startswith(id, 'renderTask')"
```

The Batch CLI supports all three clauses supported by the Batch service:

- `--select-clause [select-clause]` Return a subset of properties for each entity
- `--filter-clause [filter-clause]` Return only entities that match the specified OData expression
- `--expand-clause [expand-clause]` Obtain the entity information in a single underlying REST call. The expand clause supports only the `stats` property at this time.

For details on the three clauses and performing list queries with them, see [Query the Azure Batch service efficiently](#).

Application package management

Application packages provide a simplified way to deploy applications to the compute nodes in your pools. With the Azure CLI, you can upload application packages, manage package versions, and delete packages.

To create a new application and add a package version:

Create an application:

```
azure batch application create "resgroup001" "batchaccount001" "MyTaskApplication"
```

Add an application package:

```
azure batch application package create "resgroup001" "batchaccount001" "MyTaskApplication" "1.10-beta3" package001.zip
```

Activate the package:

```
azure batch application package activate "resgroup001" "batchaccount001" "MyTaskApplication" "1.10-beta3" zip
```

Set the **default version** for the application:

```
azure batch application set "resgroup001" "batchaccount001" "MyTaskApplication" --default-version "1.10-beta3"
```

Deploy an application package

You can specify one or more application packages for deployment when you create a new pool. When you specify a package at pool creation time, it is deployed to each node as the node joins pool. Packages are also deployed when a node is rebooted or reimaged.

Specify the `--app-package-ref` option when creating a pool to deploy an application package to the pool's nodes as they join the pool. The `--app-package-ref` option accepts a semicolon-delimited list of application ids to deploy to the compute nodes.

```
azure batch pool create --pool-id "pool001" --target-dedicated 1 --vm-size "small" --os-family "4" --app-package-ref "MyTaskApplication"
```

When you create a pool by using command-line options, you cannot currently specify *which* application package version to deploy to the compute nodes, for example "1.10-beta3". Therefore, you must first specify a default version for the application with `azure batch application set [options] --default-version <version-id>` before you create the pool (see previous section). You can, however, specify a package version for the pool if you use a [JSON file](#) instead of command line options when you create the pool.

You can find more information on application packages in [Application deployment with Azure Batch application packages](#).

IMPORTANT

You must [link an Azure Storage account](#) to your Batch account to use application packages.

Update a pool's application packages

To update the applications assigned to an existing pool, issue the `azure batch pool set` command with the `--app-package-ref` option:

```
azure batch pool set --pool-id "pool001" --app-package-ref "MyTaskApplication2"
```

To deploy the new application package to compute nodes already in an existing pool, you must restart or reimage those nodes:

```
azure batch node reboot --pool-id "pool001" --node-id "tvm-3105992504_1-20160930t164509z"
```

TIP

You can obtain a list of the nodes in a pool, along with their node ids, with `azure batch node list`.

Keep in mind that you must already have configured the application with a default version prior to deployment (`azure batch application set [options] --default-version <version-id>`).

Troubleshooting tips

This section is intended to provide you with resources to use when troubleshooting Azure CLI issues. It won't necessarily solve all problems, but it may help you narrow down the cause and point you to help resources.

- Use `-h` to get **help text** for any CLI command
- Use `-v` and `-vv` to display **verbose** command output; `-vv` is "extra" verbose and displays the actual REST requests and responses. These switches are handy for displaying full error output.
- You can view **command output as JSON** with the `--json` option. For example,

`azure batch pool show "pool001" --json` displays pool001's properties in JSON format. You can then copy and modify this output to use in a `--json-file` (see [JSON files](#) earlier in this article).

- The [Batch forum on MSDN](#) is a great help resource, and is monitored closely by Batch team members. Be sure to post your questions there if you run into issues or would like help with a specific operation.
- Not every Batch resource operation is currently supported by the Azure CLI. For example, you can't currently specify an application package *version* for a pool, only the package ID. In such cases, you may need to supply a `--json-file` for your command instead of using command-line options. Be sure to stay up-to-date with the latest CLI version to pick up future enhancements.

Next steps

- See [Application deployment with Azure Batch application packages](#) to find out how to use this feature to manage and deploy the applications you execute on Batch compute nodes.
- See [Query the Batch service efficiently](#) for more about reducing the number of items and the type of information that is returned for queries to Batch.

Log events for diagnostic evaluation and monitoring of Batch solutions

2/27/2017 • 2 min to read • [Edit Online](#)

As with many Azure services, the Batch service emits log events for certain resources during the lifetime of the resource. You can enable Azure Batch diagnostic logs to record events for resources like pools and tasks, and then use the logs for diagnostic evaluation and monitoring. Events like pool create, pool delete, task start, task complete, and others are included in Batch diagnostic logs.

NOTE

This article discusses logging events for Batch account resources themselves, not job and task output data. For details on storing the output data of your jobs and tasks, see [Persist Azure Batch job and task output](#).

Prerequisites

- [Azure Batch account](#)
- [Azure Storage account](#)

To persist Batch diagnostic logs, you must create an Azure Storage account where Azure will store the logs. You specify this Storage account when you [enable diagnostic logging](#) for your Batch account. The Storage account you specify when you enable log collection is not the same as a linked storage account referred to in the [application packages](#) and [task output persistence](#) articles.

WARNING

You are **charged** for the data stored in your Azure Storage account. This includes the diagnostic logs discussed in this article. Keep this in mind when designing your [log retention policy](#).

Enable diagnostic logging

Diagnostic logging is not enabled by default for your Batch account. You must explicitly enable diagnostic logging for each Batch account you want to monitor:

[How to enable collection of Diagnostic Logs](#)

We recommend that you read the full [Overview of Azure Diagnostic Logs](#) article to gain an understanding of not only how to enable logging, but the log categories supported by the various Azure services. For example, Azure Batch currently supports one log category: **Service Logs**.

Service Logs

Azure Batch Service Logs contain events emitted by the Azure Batch service during the lifetime of a Batch resource like a pool or task. Each event emitted by Batch is stored in the specified Storage account in JSON format. For example, this is the body of a sample **pool create event**:

```
{
  "poolId": "myPool1",
  "displayName": "Production Pool",
  "vmSize": "Small",
  "cloudServiceConfiguration": {
    "osFamily": "4",
    "targetOsVersion": "*"
  },
  "networkConfiguration": {
    "subnetId": " "
  },
  "resizeTimeout": "300000",
  "targetDedicated": 2,
  "maxTasksPerNode": 1,
  "vmFillType": "Spread",
  "enableAutoscale": false,
  "enableInterNodeCommunication": false,
  "isAutoPool": false
}
```

Each event body resides in a .json file in the specified Azure Storage account. If you want to access the logs directly, you may wish to review the [schema of Diagnostic Logs in the storage account](#).

Service Log events

The Batch service currently emits the following Service Log events. This list may not be exhaustive, since additional events may have been added since this article was last updated.

SERVICE LOG EVENTS
Pool create
Pool delete start
Pool delete complete
Pool resize start
Pool resize complete
Task start
Task complete
Task fail

Next steps

In addition to storing diagnostic log events in an Azure Storage account, you can also stream Batch Service Log events to an [Azure Event Hub](#), and send them to [Azure Log Analytics](#).

- [Stream Azure Diagnostic Logs to Event Hubs](#)

Stream Batch diagnostic events to the highly scalable data ingress service, Event Hubs. Event Hubs can ingest millions of events per second, which you can then transform and store using any real-time analytics provider.

- [Analyze Azure diagnostic logs using Log Analytics](#)

Send your diagnostic logs to Log Analytics where you can analyze them in the Operations Management Suite (OMS) portal, or export them for analysis in Power BI or Excel.

Batch and HPC solutions for large-scale computing workloads

2/27/2017 • 11 min to read • [Edit Online](#)

Azure offers efficient, scalable cloud solutions for batch and high-performance computing (HPC) - also called *Big Compute*. Learn here about Big Compute workloads and Azure's services to support them, or jump directly to [solution scenarios](#) later in this article. This article is mainly for technical decision-makers, IT managers, and independent software vendors, but other IT professionals and developers can use it to familiarize themselves with these solutions.

Organizations have large-scale computing problems: engineering design and analysis, image rendering, complex modeling, Monte Carlo simulations, financial risk calculations, and others. Azure helps organizations solve these problems with the resources, scale, and schedule they need. With Azure, organizations can:

- Create hybrid solutions, extending an on-premises HPC cluster to offload peak workloads to the cloud
- Run HPC cluster tools and workloads entirely in Azure
- Use managed and scalable Azure services such as [Batch](#) to run compute-intensive workloads without having to deploy and manage compute infrastructure

Although beyond the scope of this article, Azure also provides developers and partners a full set of capabilities, architecture choices, and development tools to build large-scale, custom Big Compute workflows. And a growing partner ecosystem is ready to help you make your Big Compute workloads productive in the Azure cloud.

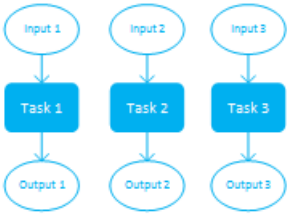
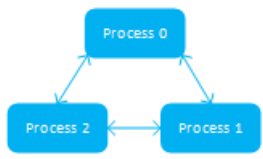
Batch and HPC applications

Unlike web applications and many line-of-business applications, batch and HPC applications have a defined beginning and end, and they can run on a schedule or on demand, sometimes for hours or longer. Most fall into two main categories: *intrinsically parallel* (sometimes called "embarrassingly parallel", because the problems they solve lend themselves to running in parallel on multiple computers or processors) and *tightly coupled*. See the following table for more about these application types. Some Azure solution approaches work better for one type or the other.

NOTE

In Batch and HPC solutions, a running instance of an application is typically called a *job*, and each job might get divided into *tasks*. And the clustered compute resources for the application are often called *compute nodes*.

TYPE	CHARACTERISTICS	EXAMPLES
------	-----------------	----------

TYPE	CHARACTERISTICS	EXAMPLES
Intrinsically parallel 	<ul style="list-style-type: none"> Individual computers run application logic independently Adding computers allows the application to scale and decrease computation time Application consists of separate executables, or is divided into a group of services invoked by a client (a service-oriented architecture, or SOA, application) 	<ul style="list-style-type: none"> Financial risk modeling Image rendering and image processing Media encoding and transcoding Monte Carlo simulations Software testing
Tightly coupled 	<ul style="list-style-type: none"> Application requires compute nodes to interact or exchange intermediate results Compute nodes may communicate using the Message Passing Interface (MPI), a common communications protocol for parallel computing The application is sensitive to network latency and bandwidth Application performance can be improved by using high-speed networking technologies such as InfiniBand and remote direct memory access (RDMA) 	<ul style="list-style-type: none"> Oil and gas reservoir modeling Engineering design and analysis, such as computational fluid dynamics Physical simulations such as car crashes and nuclear reactions Weather forecasting

Considerations for running batch and HPC applications in the cloud

You can readily migrate many applications that are designed to run in on-premises HPC clusters to Azure, or to a hybrid (cross-premises) environment. However, there may be some limitations or considerations, including:

- **Availability of cloud resources** - Depending on the type of cloud compute resources you use, you might not be able to rely on continuous machine availability while a job runs. State handling and progress check pointing are common techniques to handle possible transient failures, and more necessary when using cloud resources.
- **Data access** - Data access techniques commonly available in enterprise clusters, such as NFS, may require special configuration in the cloud. Or, you might need to adopt different data access practices and patterns for the cloud.
- **Data movement** - For applications that process large amounts of data, strategies are needed to move the data into cloud storage and to compute resources. You might need high-speed cross-premises networking such as [Azure ExpressRoute](#). Also consider legal, regulatory, or policy limitations for storing or accessing that data.
- **Licensing** - Check with the vendor of any commercial application for licensing or other restrictions for running in the cloud. Not all vendors offer pay-as-you-go licensing. You might need to plan for a licensing server in the cloud for your solution, or connect to an on-premises license server.

Big Compute or Big Data?

The dividing line between Big Compute and Big Data applications isn't always clear, and some applications may have characteristics of both. Both involve running large-scale computations, usually on clusters of computers. But the solution approaches and supporting tools can differ.

- **Big Compute** tends to involve applications that rely on CPU power and memory, such as engineering simulations, financial risk modeling, and digital rendering. The infrastructure for a Big Compute solution might include computers with specialized multicore processors to perform raw computation, and specialized, high-speed

networking hardware to connect the computers.

- **Big Data** solves data analysis problems that involve large amounts of data that can't be managed by a single computer or database management system. Examples include large volumes of web logs or other business intelligence data. Big Data tends to rely more on disk capacity and I/O performance than on CPU power. There are also specialized Big Data tools such as Apache Hadoop to manage the cluster and partition the data. (For information about Azure HDInsight and other Azure Hadoop solutions, see [Hadoop](#).)

Compute management and job scheduling

Running Batch and HPC applications often includes a *cluster manager* and a *job scheduler* to help manage clustered compute resources and allocate them to the applications that run the jobs. These functions might be accomplished by separate tools, or an integrated tool or service.

- **Cluster manager** - Provisions, releases, and administers compute resources (or compute nodes). A cluster manager might automate installation of operating system images and applications on compute nodes, scale compute resources according to demands, and monitor the performance of the nodes.
- **Job scheduler** - Specifies the resources (such as processors or memory) an application needs, and the conditions when it runs. A job scheduler maintains a queue of jobs and allocates resources to them based on an assigned priority or other characteristics.

Clustering and job scheduling tools for Windows-based and Linux-based clusters can migrate well to Azure. For example, [Microsoft HPC Pack](#), Microsoft's free compute cluster solution for Windows and Linux HPC workloads, offers several options for running in Azure. You can also build Linux clusters to run open-source tools such as Torque and SLURM. You can also bring commercial grid solutions to Azure, such as [TIBCO DataSynapse GridServer](#), [IBM Spectrum Symphony and Symphony LSF](#), and [Univa Grid Engine](#).

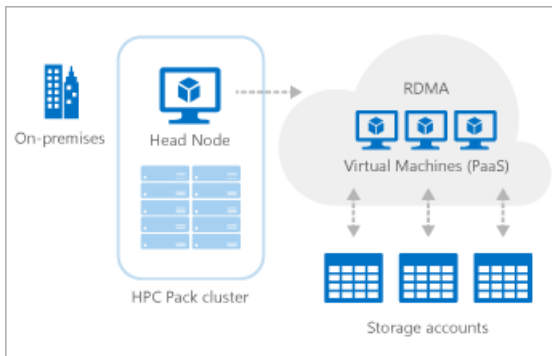
As shown in the following sections, you can also take advantage of Azure services to manage compute resources and schedule jobs without (or in addition to) traditional cluster management tools.

Scenarios

Here are three common scenarios to run Big Compute workloads in Azure by using existing HPC cluster solutions, Azure services, or a combination of the two. Key considerations for choosing each scenario are listed but aren't exhaustive. More about the available Azure services you might use in your solution is later in the article.

SCENARIO

Burst an HPC cluster to Azure



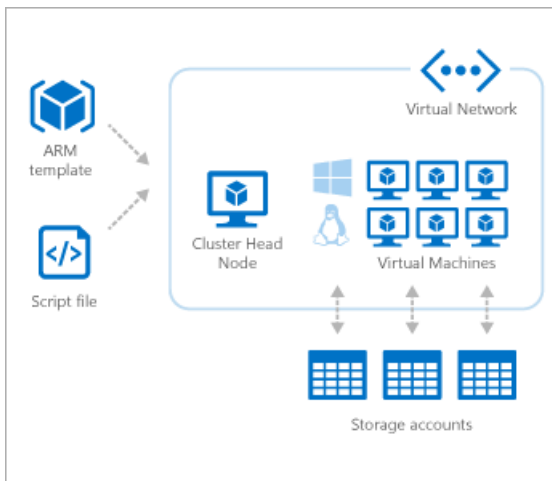
Learn more:

- [Burst to Azure worker instances with HPC Pack](#)
- [Set up a hybrid compute cluster with HPC Pack](#)
- [Burst to Azure Batch with HPC Pack](#)

WHY CHOOSE IT?

- Combine your [Microsoft HPC Pack](#) or other on-premises cluster with additional Azure resources in a hybrid solution.
- Extend your Big Compute workloads to run on Platform as a Service (PaaS) virtual machine instances (currently Windows Server only).
- Access an on-premises license server or data store by using an optional Azure virtual network

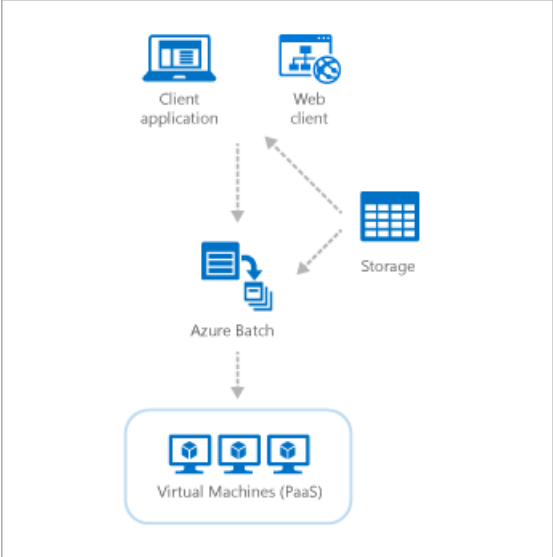
Create an HPC cluster entirely in Azure



Learn more:

- [HPC cluster solutions in Azure](#)

- Quickly and consistently deploy your applications and cluster tools on standard or custom Windows or Linux infrastructure as a service (IaaS) virtual machines.
- Run various Big Compute workloads by using the job scheduling solution of your choice.
- Use additional Azure services including networking and storage to create complete cloud-based solutions.

SCENARIO	WHY CHOOSE IT?
<p>Scale out a parallel application to Azure</p>  <p>Learn more:</p> <ul style="list-style-type: none"> • Basics of Azure Batch • Get started with the Azure Batch library for .NET 	<ul style="list-style-type: none"> • Develop with Azure Batch to scale out various Big Compute workloads to run on pools of Windows or Linux virtual machines. • Use an Azure platform service to manage deployment and autoscaling of virtual machines, job scheduling, disaster recovery, data movement, dependency management, and application deployment.

Azure services for Big Compute

Here is more about the compute, data, networking, and related services you can combine for Big Compute solutions and workflows. For in-depth guidance on Azure services, see the Azure services [documentation](#). The [scenarios](#) earlier in this article show just some ways of using these services.

NOTE

Azure regularly introduces new services that could be useful for your scenario. If you have questions, contact an [Azure partner](#) or email bigcompute@microsoft.com.

Compute services

Azure compute services are the core of a Big Compute solution, and the different compute services offer advantages for different scenarios. At a basic level, these services offer different modes for applications to run on virtual machine-based compute instances that Azure provides using Windows Server Hyper-V technology. These instances can run standard and custom Linux and Windows operating systems and tools. Azure gives you a choice of [instance sizes](#) with different configurations of CPU cores, memory, disk capacity, and other characteristics. Depending on your needs, you can scale the instances to thousands of cores and then scale down when you need fewer resources.

NOTE

Take advantage of the Azure [compute-intensive instances such as the H-series](#) to improve the performance and scalability of HPC workloads. These instances also support parallel MPI applications that require a low-latency and high-throughput application network. Also available are [N-series](#) VMs with NVIDIA GPUs to expand the range of computing and visualization scenarios in Azure.

SERVICE	DESCRIPTION
Virtual machines	<ul style="list-style-type: none"> • Provide compute infrastructure as a service (IaaS) using Microsoft Hyper-V technology • Enable you to flexibly provision and manage persistent cloud computers from standard Windows Server or Linux images from the Azure Marketplace, or images and data disks you supply • Can be deployed and managed as VM Scale Sets to build large-scale services from identical virtual machines, with autoscaling to increase or decrease capacity automatically • Run on-premises compute cluster tools and applications entirely in the cloud
Cloud services	<ul style="list-style-type: none"> • Can run Big Compute applications in worker role instances, which are virtual machines running a version of Windows Server and are managed entirely by Azure • Enable scalable, reliable applications with low administrative overhead, running in a platform as a service (PaaS) model • May require additional tools or development to integrate with on-premises HPC cluster solutions
Batch	<ul style="list-style-type: none"> • Runs large-scale parallel and batch workloads in a fully managed service • Provides job scheduling and autoscaling of a managed pool of virtual machines • Allows developers to build and run applications as a service or cloud-enable existing applications

Storage services

A Big Compute solution typically operates on a set of input data, and generates data for its results. Some of the Azure storage services used in Big Compute solutions include:

- [Blob, table, and queue storage](#) - Manage large amounts of unstructured data, NoSQL data, and messages for workflow and communication, respectively. For example, you might use blob storage for large technical data sets, or for the input images or media files your application processes. You might use queues for asynchronous communication in a solution. See [Introduction to Microsoft Azure Storage](#).
- [Azure File storage](#) - Shares common files and data in Azure using the standard SMB protocol, which is needed for some HPC cluster solutions.
- [Data Lake Store](#) - Provides a hyperscale Apache Hadoop Distributed File System for the cloud, useful for batch, real-time, and interactive analytics.

Data and analysis services

Some Big Compute scenarios involve large-scale data flows, or generate data that needs further processing or analysis. Azure offers several data and analysis services, including:

- [Data Factory](#) - Builds data-driven workflows (pipelines) that join, aggregate, and transform data from on-premises, cloud-based, and Internet data stores.
- [SQL Database](#) - Provides the key features of a Microsoft SQL Server relational database management system in a managed service.

- [HDInsight](#) - Deploys and provisions Windows Server or Linux-based Apache Hadoop clusters in the cloud to manage, analyze, and report on big data.
- [Machine Learning](#) - Helps you create, test, operate, and manage predictive analytic solutions in a fully managed service.

Additional services

Your Big Compute solution might need other Azure services to connect to resources on-premises or in other environments. Examples include:

- [Virtual Network](#) - Creates a logically isolated section in Azure to connect Azure resources to each other or to your on-premises data center. With a cross-premises virtual network, Big Compute applications can access on-premises data, Active Directory services, and license servers
- [ExpressRoute](#) - Creates a private connection between Microsoft data centers and infrastructure that's on-premises or in a co-location environment. ExpressRoute provides higher security, more reliability, faster speeds, and lower latencies than typical connections over the Internet.
- [Service Bus](#) - Provides several mechanisms for applications to communicate or exchange data, whether they are located on Azure, on another cloud platform, or in a data center.

Next steps

- See [Technical Resources for Batch and HPC](#) to find technical guidance to build your solution.
- Discuss your Azure options with partners including Cycle Computing, Rescale, and UberCloud.
- Read about Azure Big Compute solutions delivered by [Towers Watson](#), [Altair](#), [ANSYS](#), and [d3VIEW](#).
- For the latest announcements, see the [Microsoft HPC and Batch team blog](#) and the [Azure blog](#).

Big Compute in Azure: Technical resources for batch and high-performance computing

3/27/2017 • 2 min to read • [Edit Online](#)

This is a guide to technical resources to help you run your large-scale parallel, batch, and high-performance computing (HPC) workloads in Azure. Extend your existing batch or HPC workloads to the Azure cloud, or build new Big Compute solutions using a range of Azure services.

Solutions options

Learn about Big Compute options in Azure, and choose the right approach for your workload and business need.

- [Batch and HPC solutions](#)
- [Video: Big Compute in the cloud with Azure and HPC](#)

Azure Batch

[Batch](#) is a platform service that makes it easy to cloud-enable your Linux and Windows applications and run jobs without setting up and managing a cluster and job scheduler. Use the SDK to integrate client applications with Azure Batch through various languages, stage data to Azure, and build job execution pipelines.

- [Documentation](#)
- [.NET, Python, Node.js, Java, and REST API reference](#)
- [Batch management .NET library](#) reference
- Tutorials: Get started with [Azure Batch library for .NET](#) and [Batch Python client](#)
- [Batch forum](#)
- [Batch videos](#)

HPC cluster solutions

Deploy or extend your existing Windows or Linux HPC cluster to Azure to run your compute intensive workloads.

Microsoft HPC Pack

HPC Pack is Microsoft's free HPC solution built on Microsoft Azure and Windows Server technologies, capable of running Windows and Linux HPC workloads.

- [Download HPC Pack 2016](#)
- [Download HPC Pack 2012 R2 Update 3](#)
- [Documentation](#)
- HPC Pack cluster options in Azure: [Linux](#) and [Windows](#)
- [Burst to Azure worker instances with HPC Pack](#)
- [Burst to Azure Batch with HPC Pack](#)
- [Windows HPC forums](#)

Linux and OSS cluster solutions

Use these Azure templates to deploy Linux HPC clusters.

- [Spin up a SLURM cluster](#) and [blog post](#)
- [Spin up a Torque cluster](#)

- [Compute grid templates with PBS Professional](#)

HPC storage

- [Parallel file systems for HPC storage on Azure](#)
- [Intel Cloud Edition for Lustre Software - Eval](#)
- [BeeGFS on CentOS 7.2 template](#)

Microsoft MPI

[Microsoft MPI](#) (MS-MPI) is a Microsoft implementation of the Message Passing Interface standard for developing and running parallel applications on the Windows platform.

- [Download MS-MPI](#)
- [MS-MPI reference](#)
- [MPI forum](#)

Compute-intensive instances

Azure offers a [range of VM sizes](#), including [compute-intensive H-series](#) instances capable of connecting to a back-end RDMA network, to run your Linux and Windows HPC workloads.

- [Set up a Linux RDMA cluster to run MPI applications](#)
- [Set up a Windows RDMA cluster with Microsoft HPC Pack to run MPI applications](#)

For GPU-intensive workloads, check out [NC and NV sizes](#).

Samples and demos

- [Azure Batch C# and Python code samples](#)
- [Batch Shipyard](#) toolkit for easy deployment of batch-style Dockerized workloads to Azure Batch
- [doAzureParallel](#) R package, built on top of Azure Batch
- [Test drive SUSE Linux Enterprise Server for HPC](#)

Related Azure services

- [Data Factory](#)
- [Machine Learning](#)
- [HDInsight](#)
- [Virtual Machines](#)
- [Virtual Machine Scale Sets](#)
- [Cloud Services](#)
- [App Service](#)
- [Media Services](#)
- [Functions](#)

Architecture blueprints

- [HPC and data orchestration using Azure Batch and Azure Data Factory](#) (PDF) and [article](#)

Industry solutions

- [Banking and capital markets](#)

- [Engineering simulations](#)

Customer stories

- [ANEO](#)
- [d3View](#)
- [Ludwig Institute of Cancer Research](#)
- [Microsoft Research](#)
- [Milliman](#)
- [Mitsubishi UFJ Securities International](#)
- [Schlumberger](#)
- [Towers Watson](#)
- [UberCloud](#)

Next steps

- For the latest announcements, see the [Microsoft HPC and Batch team blog](#) and the [Azure blog](#).
- Also see [what's new in Batch](#) or subscribe to the [RSS feed](#).