

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

06/28/2017 • 26 minutes to read • Contributors       all

In this article

[Prerequisites](#)

[DotNetTutorial sample project overview](#)

[Build the DotNetTutorial sample project](#)

[Step 1: Create Storage containers](#)

[Step 2: Upload task application and data files](#)

[Step 3: Create Batch pool](#)

[Step 4: Create Batch job](#)

[Step 5: Add tasks to job](#)

[Step 6: Monitor tasks](#)

[Step 7: Download task output](#)

[Step 8: Delete containers](#)

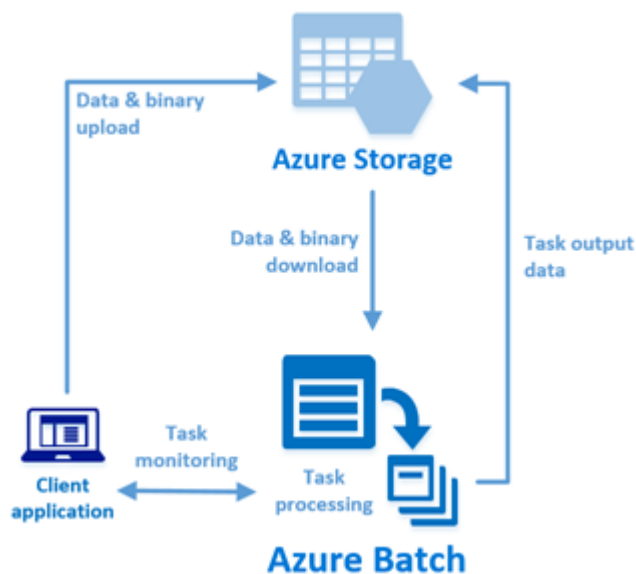
[Step 9: Delete the job and the pool](#)

[Run the DotNetTutorial sample](#)

[Next steps](#)

.NET

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-samples\CSharp\ArticleProjects\DotNetTutorial
```

BatchLabs (optional)

[BatchLabs](#) is a free, rich-featured, standalone client tool to help create, debug, and monitor Azure Batch applications. While not required to complete this tutorial, it can be useful while developing and debugging your Batch solutions.

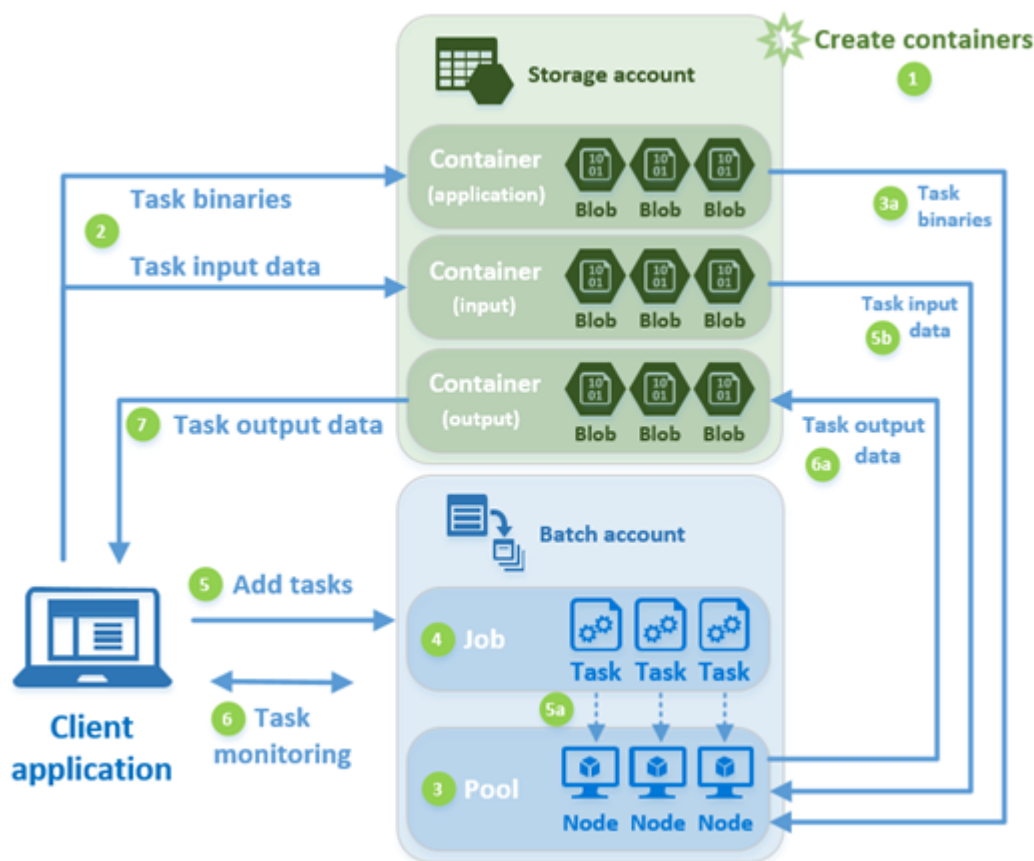
DotNetTutorial sample project overview

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 `DotNetTutorial.sln` solution file. Or open it from within Visual Studio by using the **File > Open > Project/Solution** menu.

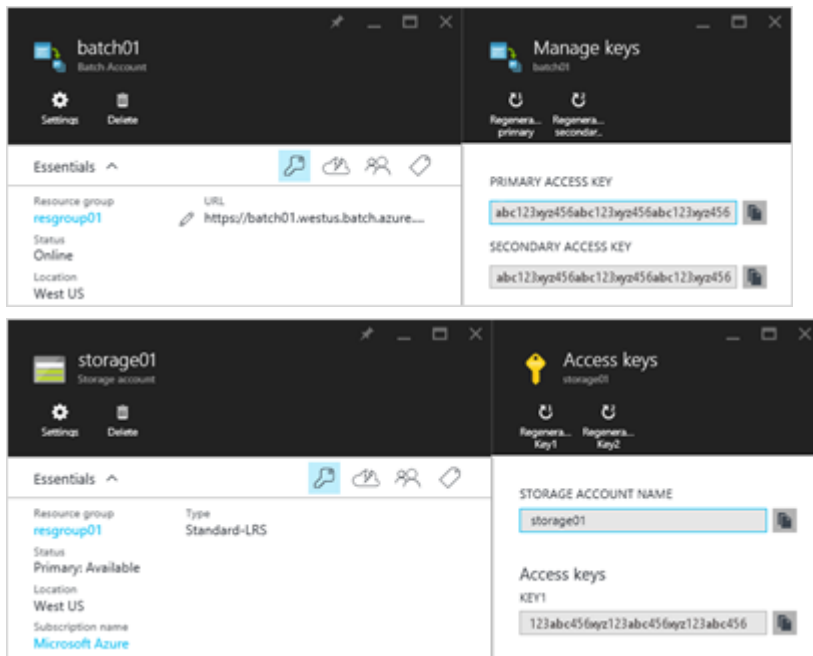
Open `Program.cs` within the *DotNetTutorial* project. Then add your credentials as specified near the top of the file:

C#	 Copy
<pre>// 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 = "";</pre>	

ⓘ 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.

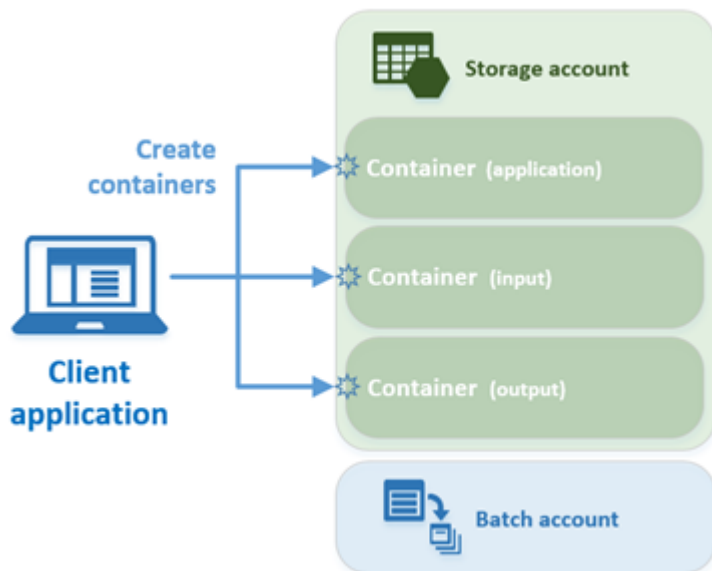
Tip

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](#):

C#

Copy

```
// 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.

C#

 Copy

```
// 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);
```

C#

 Copy

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

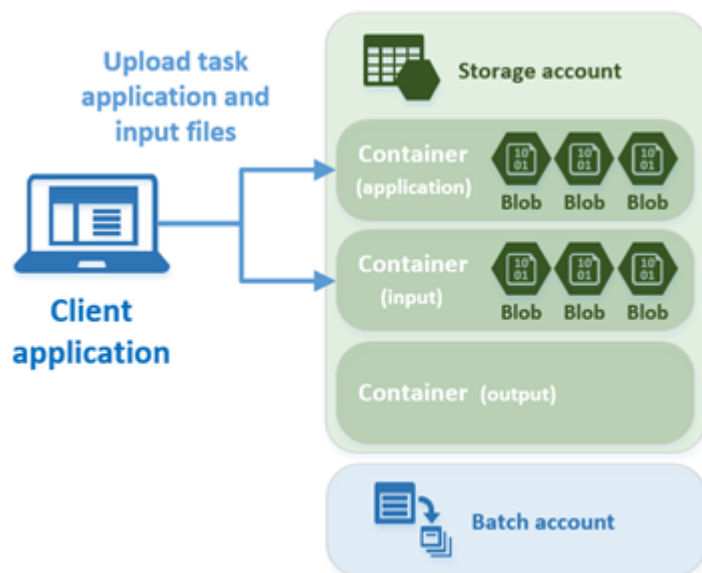
    if (await container.CreateIfNotExistsAsync())
    {
        Console.WriteLine("Container [{0}] created.", containerName);
    }
    else
    {
        Console.WriteLine("Container [{0}] exists, skipping creation.",
            containerName);
    }
}
```

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.

C#

Copy

```
// 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 *filePaths* 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.

C#

 Copy

```
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.

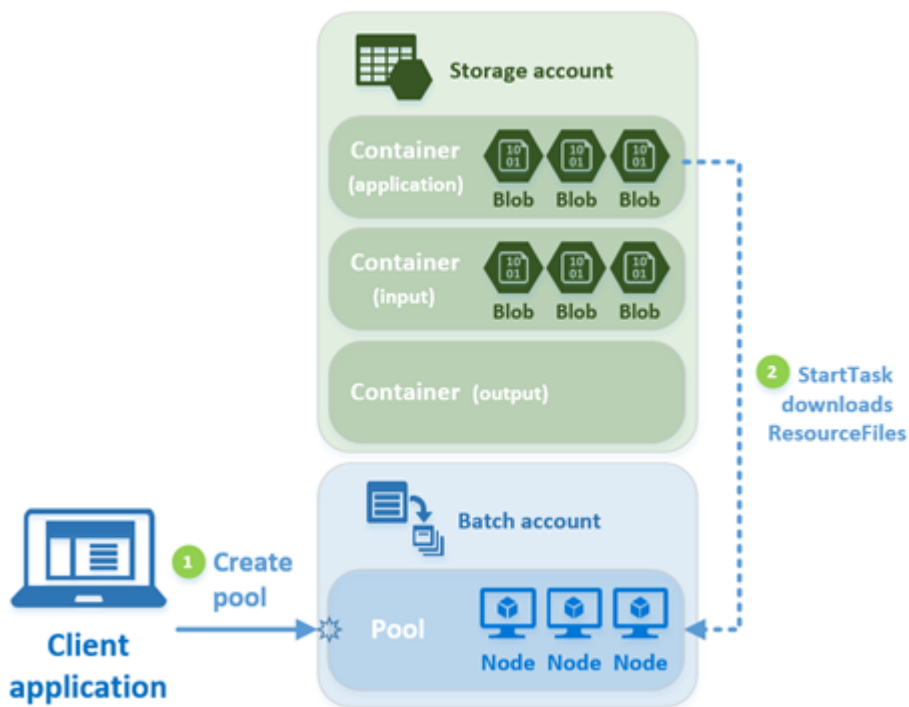
- **Blob shared access signatures:** The pool's StartTask in DotNetTutorial uses blob shared access signatures when it downloads the application binaries and input data files from Storage (see Step #3 below). The `UploadFileToContainerAsync` method in DotNetTutorial's `Program.cs` contains the code that 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 uploading the application and data files to the Storage account with Azure Storage APIs, *DotNetTutorial* begins making calls to the Batch service with APIs provided by the Batch .NET library. The code first creates a [BatchClient](#):

C#

Copy

```
BatchSharedKeyCredentials cred = new BatchSharedKeyCredentials(  
    BatchAccountUrl,  
    BatchAccountName,
```

```
BatchAccountKey));

using (BatchClient batchClient = BatchClient.Open(cred))
{
    ...
}
```

Next, the sample creates a pool of compute nodes in the Batch account with a call to

`CreatePoolIfNotExistsAsync`. `CreatePoolIfNotExistsAsync` uses the

`BatchClient.PoolOperations.CreatePool` method to create a new pool in the Batch service:

C#

Copy

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

        // Create the unbound pool. Until we call CloudPool.Commit() or CommitAsync(), no
        // Batch service. This CloudPool instance is therefore considered "unbound," and w
        pool = batchClient.PoolOperations.CreatePool(
            poolId: poolId,
            targetDedicatedComputeNodes: 3, //
            virtualMachineSize: "small", //
            cloudServiceConfiguration: new CloudServiceConfiguration(osFamily: "4")); //

        // Create and assign the StartTask that will be executed when compute nodes join t
        // In this case, we copy the StartTask's resource files (that will be automaticall
        // to the node by the StartTask) into the shared directory that all tasks will hav
        pool.StartTask = new StartTask
        {
            // Specify a command line for the StartTask that copies the task application f
            // node's shared directory. Every compute node in a Batch pool is configured w
            // of pre-defined environment variables that can be referenced by commands or
            // run by tasks.

            // Since a successful execution of robocopy can return a non-zero exit code (e
            // more files were successfully copied) we need to manually exit with a 0 for
            // StartTask execution success.
            CommandLine = "cmd /c (robocopy %AZ_BATCH_TASK_WORKING_DIR% %AZ_BATCH_NODE_SHA
            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.PoolExists)
{
    Console.WriteLine("The pool {0} already existed when we tried to create it", poolName);
}
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](#).

You can also create pools of compute nodes that are Azure Virtual Machines (VMs) by specifying the [VirtualMachineConfiguration](#) for your pool. You can create a pool of VM compute nodes from either Windows or [Linux images](#). The source for your VM images can be either:

- The [Azure Virtual Machines Marketplace](#), which provides both Windows and Linux images that are ready-to-use.
- A custom image that you prepare and provide. For more details about custom images, see [Develop large-scale parallel compute solutions with Batch](#).

ⓘ Important

You are charged for compute resources in Batch. To minimize costs, you can lower `targetDedicatedComputeNodes` 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

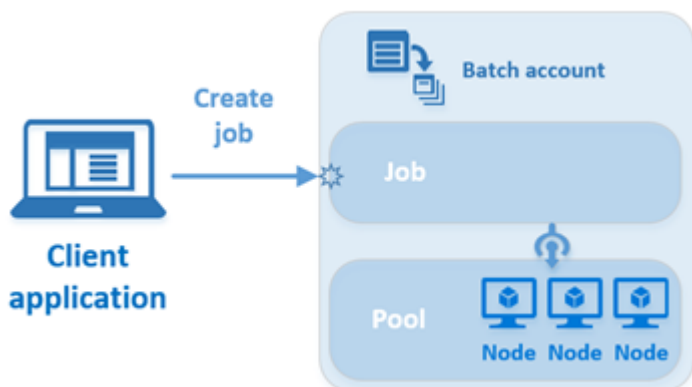
The **application packages** feature of Azure Batch provides another way to get your application onto the compute nodes in a pool. See [Deploy applications to compute nodes with Batch application packages](#) for details.

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.

C#

 Copy

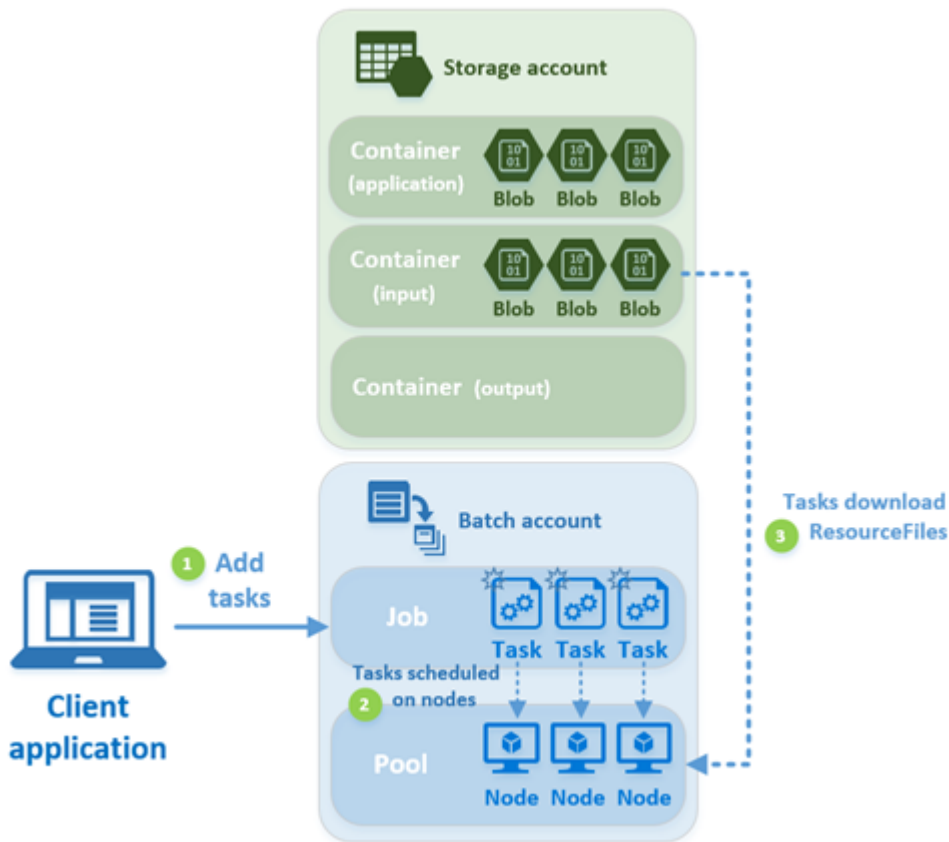
```
private static async Task CreateJobAsync(
    BatchClient batchClient,
    string jobId,
    string poolId)
{
    Console.WriteLine("Creating job [{0}]...", jobId);

    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.

C#

Copy

```
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:

```
C# Copy

// NOTE: From project TaskApplication Program.cs

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

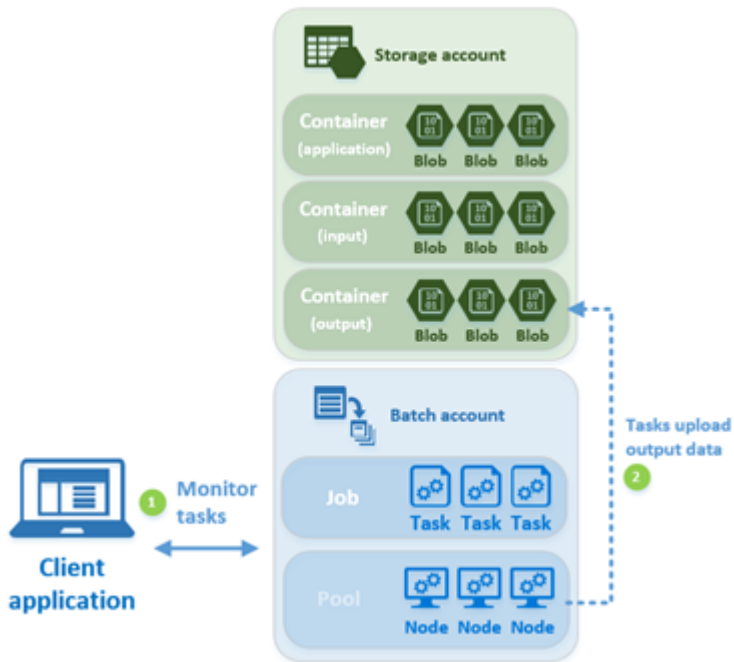
    // Obtain a reference to the container using the SAS URI.
    CloudBlobContainer container = new CloudBlobContainer(new Uri(containerSas));

    // 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:

C#

```
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 w

    // 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);

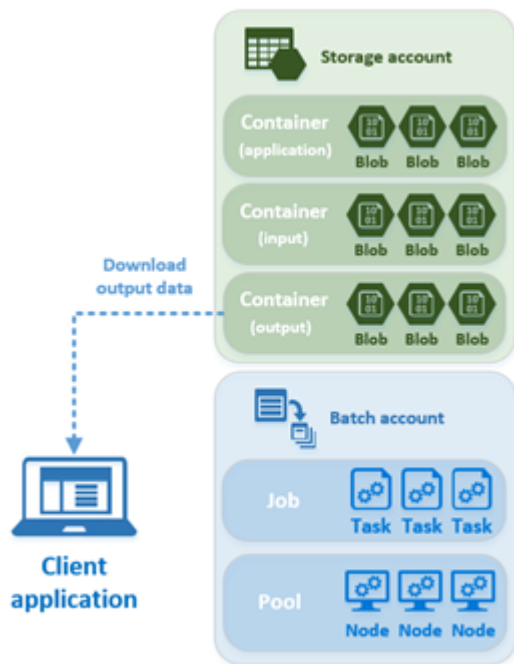
    // All tasks have reached the "Completed" state, however, this does not
    // guarantee all tasks completed successfully. Here we further check each task's
    // ExecutionInfo property to ensure that it did not encounter a failure
    // or return a non-zero exit code.

    // 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.Result == TaskExecutionResult.Failure)  
    {  
        // A task with failure information set indicates there was a problem with the  
        // the task's state can be "Completed," yet still have encountered a failure.  
  
        allTasksSuccessful = false;  
  
        Console.WriteLine("WARNING: Task [{0}] encountered a failure: {1}", task.Id, t  
        if (task.ExecutionInformation.ExitCode != 0)  
        {  
            // A non-zero exit code may indicate that the application executed by the  
            // during execution. As not every application returns non-zero on failure  
            // your implementation of error checking may differ from this example.  
  
            Console.WriteLine("WARNING: Task [{0}] returned a non-zero exit code - thi  
        }  
    }  
}  
  
if (allTasksSuccessful)  
{  
    Console.WriteLine("Success! All tasks completed successfully within the specified  
}  
  
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` :

C#

Copy

```
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`:

C#

Copy

```
// 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`:

C#

Copy

```
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);  
    }  
}
```