Migrating from Azure Batch Apps to Azure Batch

Introduction

Azure Batch is a fully-managed cloud service that enables you to run large-scale parallel and high performance computing (HPC) applications efficiently in the cloud. 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.

When Batch first became available on Azure, a number of additional capabilities like application packages, job splitter, and task processor, were made available as part of a preview service named Batch Apps. The Batch Apps preview functionality is now available in Batch, so you can now migrate any workloads that are using Batch Apps to Batch, by using the updated Batch API. This document explains how to perform that migration.

> **Important**
> The preview of Batch Apps will be discontinued on September 30, 2016. To avoid downtime, you must complete the migration of your Batch Apps workloads to Batch before this date.

Architectural Overview

Although Batch provides the same core functionality of highly-scalable cloud computing as Batch Apps does, Batch has a different programming model. Your application that uses Batch Apps will need changes in the following areas, each of which is discussed in more detail later in this document:

- Job model: Batch jobs are not marked as 'succeeded' or 'failed,' only as 'completed'
- Job types: Batch does not have the notion of a 'job type'; it depends on client programs specifying command lines.
- Job and task parameters: Because Batch doesn't use job types, it doesn't directly support parameters; instead you'll need to pass parameters via files or environment variables, or on task command lines.
- Cloud assembly: Batch does not have 'cloud assemblies,' only executable programs; you can port your cloud assembly code using a set of Visual Studio templates, as discussed later in this document.
- Application images: Although you can use your existing application image files, the way you manage and deploy them changes in Batch.
- File management: Batch does not provide an API for managing input and output files; you will need to invoke Azure Storage directly. This impacts both clients (uploading input files, listing and downloading outputs) and task code (persisting outputs).
- Stages: in Batch you define processing order within a job as dependencies between tasks, rather than as stages.
- Merge task: Batch doesn't have a merge task; any merge needs to be coded explicitly.
- Monitoring and diagnostics: Batch doesn't provide integrated logging, instead processor output is exposed directly via files that capture standard output and standard error.

The recommended model to follow when migration your solution from Batch Apps to Batch is:

1. Port your task processor code to a task processing wrapper program.
2. Port your job splitter code to a job manager program.
3. Deploy the task processor and job manager as an application package.
4. Port your client code to the Azure Batch client SDK.

This is not the only model you can follow, but it adheres most closely to the Batch Apps architecture, and will therefore likely require the least amount of effort.

Task Processor impact

The differences highlighted earlier in this document mean that the task processor code needs to change as follows – again, we will discuss each item in more detail below.

1. Bring your code across to an executable, using the Batch Visual Studio templates.
2. Update your code to use Azure Storage mechanisms for saving output files – the Batch file conventions library provides a convenient wrapper which mimics Batch Apps storage and lookup conventions.
3. Update your code to return a process exit code instead of a success enum.
4. Change your logging code to write to standard out and standard error.
5. If you use the Batch Apps merge task, create a separate task processor for that.

Job Splitter impact

For job splitters, the main changes are as follows:

1. Bring your code across to an executable, using the Batch Visual Studio templates.
2. Update your task creation code to return Batch CloudTask objects.
3. Change task input file references to resource files.
4. If you use Batch Apps job stages, rewrite these as task dependencies.
5. Change task parameter code to pass parameters via a file.
6. If you use Batch Apps ignorable failures, set up task exit conditions for these.
7. If you use the Batch Apps merge task, emit it explicitly and set up appropriate inputs.

Client impact

For client code, the main changes are as follows:

1. Change job creation code to use Batch client classes.
2. Write code to manage the upload of input files and their notification to the job manager.
3. Write job parameters to a file and pass this as a resource file to the job manager.
4. Update how you detect job and task completion and success.
5. Change job and task output retrieval code to use Azure Storage mechanisms.
6. Set up the desired auto job completion behaviour.

Prerequisites

To migrate your existing solution to Batch, you will need the following:

- A computer with Visual Studio 2015, or newer, already installed on it.
- The zip file for your current Batch Apps application image.

- The code for the different pieces of your current Batch Apps solution (e.g., task processor, job splitter, client, dependencies, etc.)

Step 1: Preparation

1.1 Set up linked storage on your Batch account

All Batch Apps accounts are automatically linked to a storage account. Batch accounts are not automatically linked, so you must set this up in the Azure portal.

1. Open the Azure portal (https://portal.azure.com).
2. Click **Browse**, and then click **Storage accounts**.
3. Click **Add**, and then follow the steps to create a storage account in the *same region and subscription as your Batch account*. You must choose **General Purpose** as the account kind and **Standard** performance.
4. After the storage account is created, click **Browse**, click **Batch Accounts**, and then select your Batch account.
5. Click **Settings**.
6. Click **Storage Account**.
7. Click **Storage Account** again.
8. Select the storage account you created earlier. If the storage account you created is not listed, close the list and wait a few minutes before trying again.
9. After a few seconds, the storage account will be listed as being linked to the Batch account.

1.2 Install your application image as an application package

In Batch Apps, you package external programs that you want to call from your task processor as *application images*. In Batch, application images are known as *application packages*. You can reuse your existing application image, but you will need to upload it into your Batch account.

1. Open the Azure portal and navigate to your Batch account.
2. Click **Applications**.
3. Click **Add**.
4. Type an application ID and version, and select the application image zip file.
5. Click **OK** to upload the application image as an application package and make it available in your Batch account.

1.3 Install the Visual Studio templates

Batch provides Visual Studio templates to get you started writing your job manager and task processor. To install the templates, follow these steps:

1. Open Visual Studio 2015.
2. On the **Tools** menu, click **Extensions and Updates**.
3. On the navigation pane, click **Online**.
4. Search for “Azure Batch Templates”.
5. In the results list, look for “Microsoft Azure Batch Project Templates”.
6. Select it and click **Download**, and then follow the steps to install the templates.

1.4 Choose an ID for your task code

Later on, you will deploy your task code (the equivalent of your Batch Apps cloud assembly) as an application package. You'll need to know the package ID during development, so choose it now and note it down.

Step 2: Port the Task Processor

2.1 Create a Visual Studio project for the task processor

In Batch Apps, your task processor code lived in a DLL created using the cloud assembly template. In Batch, your task processor code will live in an EXE created using the Batch Task Processor template.

We recommend creating a solution that can contain your job manager as well as your task processor, as this makes it easier to share code. To create this solution, follow these steps:

1. Open Visual Studio 2015 and under **File**, click **New**, and then click **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**.
5. Right-click the solution in Solution Explorer and then click **Add**, and **New Project**.
6. Under **Visual C#**, click **Cloud**, and then click **Azure Batch Task Processor**.
7. Type a name that describes your application and identifies this project as the task processor (e.g. "LitwareTaskProcessor").
8. To create the project, click **OK**.
9. 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.

2.2 Copy your current code

Copy your existing `RunExternalTaskProcess()` code into the generated `Run()` method of the task processor code in Visual Studio, replacing the sample code generated from the template. If your existing code depends on other classes or methods, copy them to the new project too. Remember to add any references your existing code needs, *except* the Batch Apps Cloud SDK – that is replaced by the Visual Studio template and the Batch SDK.

2.3 Update task parameter references

In Batch Apps, task parameters were accessed via a Parameters property on the ITask interface. In Batch, with the task processor template, they are provided as a field named `_parameters`. Change all references to `task.Parameters` to `_parameters`.

2.4 Change the external process invocation code

In Batch Apps, you used the ExecutablesPath to access programs from the application image. In Batch, there can be multiple application packages, so you need to construct program paths using application package environment variables. For example, if the program you want to invoke is named *convert.exe*, and is in an application package with ID *imagemagick* and version *7.0*, you'd construct the command line as follows:

```
var packageFolder = Environment.GetEnvironmentVariable("AZ_BATCH_APP_PACKAGE_IMAGEMAGICK#7.0");
var programPath = Path.Combine(packageFolder, "convert.exe");
```

If you use the `ExternalProcess` helper class to run the program, change it to call the .NET `Process.Start` method directly. Batch does not include an equivalent of `ExternalProcess`. `ExternalProcess` also handles waiting for the external process, and transforms exit codes into exceptions, so you need to modify your code to wait for process exit and move your success and failure handling code under a `Process.ExitCode` check, as shown here:

```
using (var process = Process.Start(programPath, arguments))
{
  process.WaitForExit();
  if (process.ExitCode == 0)
  {
    // success code
  }
  else
  {
    // failure code – would have been in an exception handler if using ExternalProcess
  }
}
```

2.5 Change the output file list to 'save' statements

In Batch Apps, the task processor could return a list of files to be persisted, and the Batch Apps framework saves them to Azure blob storage automatically. In Batch, if you want to persist files, you must do so explicitly. For each file in your task processor's Outputs list, call the `SaveAsync` method for the task output storage, as shown here:

```
// Batch Apps
return TaskProcessResult.FromExternalProcessResult(
    externalProcessResult,
    "results.csv",
    "result_graph.jpg"
);

// Batch
var taskOutputStorage = new TaskOutputStorage(
    _configuration.StorageAccount,
    _configuration.JobId,
    _configuration.TaskId
);
await taskOutputStorage.SaveAsync(TaskOutputKind.TaskOutput, "results.csv");
await taskOutputStorage.SaveAsync(TaskOutputKind.TaskOutput, "result_graph.jpg");
```

2.6 Change the success enum to an exit code

In Batch Apps, you would return a value from the `TaskProcessSuccess` enumeration to indicate whether the task had succeeded or failed. Change this to return a process exit code – 0 for success, non-zero for failure. In many cases you can just return the exit code of the external program you invoked from the task processor, as shown here:

```
// Batch Apps
return new TaskProcessResult { Success = TaskProcessSuccess.Succeeded };

// Batch
return 0;  // or process.ExitCode
```

By default, if you turn on task failure detection for the job (see 5.2 Update job creation code, later in this document), then all task failures result in job termination, just as in Batch Apps. If you want a particular task failure to not result in job termination, then you must choose an exit code to represent 'ignorable' failure, as shown here:

```
const int IgnorableFailure = 301;
//
// ... task processing code ...
//
return IgnorableFailure;
```

Later, in the job manager, you will specify appropriate exit conditions on the task to cause this exit code to be ignored.

2.7 Update any logging statements

In Batch Apps, you could log progress or diagnostic information to the job log using the `ILog` interface. Batch doesn't provide a built-in logging facility, but it does record standard output and error to files named stdout.txt and stderr.txt. Change your `Log` statements to `Console.Out.WriteLine` statements. You may want to change some statements to `Console.Error.WriteLine` instead, but this can make it harder for you to view a consolidated log, as some messages will then end up in stdout.txt and others in stderr.txt.

2.8 Save the logging files

In Batch Apps, logs were automatically saved and displayed in the Batch Apps preview portal. In Batch, you must save the stdout.txt and stderr.txt files to blob storage if you want them to be available after the compute node is torn down. Use the `TaskOutputStorage.SaveTrackedAsync` method to persist these files.

2.9 Port your merge task code

The Batch Apps task processor framework provided a separate entry point for the merge task. Since Batch task processors are just executables, this is not possible in Batch. If your jobs contain a merge task, you can do one of the following:

- Create a separate task processor program to perform the merge; or
- Use a command line switch or task parameter to indicate that the task processor is being called to perform a merge, and have your `Run` implementation branch on this.

2.10 Make a note of the task processor program name

In Batch Apps, tasks were automatically associated with a task processor through the job type. In Batch, tasks must be created with a reference to the task processor command line. You'll therefore need to know the name of the task processor program for when you port your job splitter code. Check the assembly name in the Visual Studio project properties page and note it down for use in Step 3.

Step 3: Port the Job Splitter

3.1 Create a Visual Studio project for the job manager

In Batch Apps, your job splitter code lived in a DLL created using the cloud assembly template. In Batch, your job manager code lives in an EXE created using the Batch Job Manager template.

We recommend putting your job manager in the same Visual Studio solution as your task processor, as this makes it easier to share code. To create this solution, follow these steps:

1. Open your existing solution in Visual Studio 2015.
2. Right-click the solution in Solution Explorer and then click **Add**, and **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.

3.2 Copy your current code

Copy your existing `Split()` code into the generated `Split()` method, replacing the sample code generated from the template. As with the task processor, also copy any code or references that your `Split()` method needs, *except* for the Batch Apps Cloud SDK reference.

3.3 Update job parameter references

In Batch Apps, job parameters were accessed via a `Parameters` property on the `IJob` interface. In the task processor template, they are provided as a field named `_parameters`. Change all references to `job.Parameters` to `_parameters`.

3.4 Change TaskSpecifiers to CloudTasks

Your `Split` method returns a sequence of tasks. In Batch Apps, these were represented by the `TaskSpecifier` type. In Batch, they are represented by the `CloudTask` type. Although you can use your existing logic for choosing how to divide work up into tasks, `CloudTask` has different properties from `TaskSpecifier` and you will need to configure it accordingly.

Batch Apps task IDs were integers, and were automatically inferred if you didn't specify them; Batch task IDs are strings, and must be specified explicitly. Add a counter to your code and call `ToString()` on it to derive an ID, as shown here:

```
// Batch Apps
foreach (var unitOfWork in unitsOfWork)
{
    yield return new TaskSpecifier { ... };
}

// Batch
int taskId = 0;
foreach (var unitOfWork in unitsOfWork)
{
    taskId++;
    yield return new CloudTask(id: taskId.ToString(), ...);
}
```

Tasks in Batch Appswere mapped to their task processors automatically; tasks in Batch do not do this. You need to set the `CloudTask.CommandLine` to the task processor command line. In order to access the task processor program path, you should write this as follows:

```
var taskCommandLine = @"cmd /c %AZ_BATCH_APP_PACKAGE_{TaskPkgId}#1.0%\{taskprocessor}.exe";
```

where *{TaskPkgId}* is the task code package ID you chose in step 1.4 and *{taskprocessor}* is the program name you noted in step 2.10.

Batch Apps downloaded cloud assemblies and application images automatically; Batch requires these to be configured on tasks. Set `CloudTask.ApplicationPackageReferences` to a list containing your external program application package (from step 1.2) and your task code application package (from step 1.4), as shown here:

```
task.ApplicationPackageReferences = new [] {
  new ApplicationPackageReference { ApplicationId = "{ExtProgPkgId}", Version = "{ExtProgPkgVer}" },
  new ApplicationPackageReference { ApplicationId = "{TaskPkgId}", Version = "1.0" },
};
```

where *{ExtProgPkgId}* is the application id you chose in step 1.2, *{ExtProgPkgVer}* is the package version you chose in step 1.2, and *{TaskPkgId}* is the task code package ID you chose in step 1.4.

Batch Apps had a task dependency model based on stages. The dependency model in Batch is based on task-to-task dependencies. If your existing code uses stages, you will need to convert the stages into task ID ranges, and reference those in the `CloudTask.DependsOn` property, as shown here:

```
// Batch Apps
taskSpecifier.Stage = 2;

// Batch
task.DependsOn = TaskDependencies.OnIdRange(previousStageStartId, previousStageEndId);
```

Batch Apps had a built-in notion of task exit behaviors. Batch requires you to define any non-default behaviors explicitly. This is only required if you are using 'ignorable' failures (that is, task failures that do not terminate the job; in this case, list ignorable failure exit codes on the task, as shown here:

```
ExitOptions ignorableFailure = new ExitOptions {
  JobAction = JobAction.None,
};

task.ExitConditions = new ExitConditions {
  ExitCodes = new [] { new ExitCodeMapping(/* ignorable failure exit code */, ignorableFailure) },
};
```

Additionally, if you wish to be able to reprocess the job after a failure, you must set the default action on failure to disable the job rather than terminate it, as shown here:

```
task.ExitConditions = new ExitConditions {
  Default = new ExitOptions { JobAction = JobAction.Disable },
};
```

3.5 Update how parameters are passed to tasks

In Batch Apps, the TaskSpecifier object had a `Parameters` collection which you could use to configure each task as you created it. This collection doesn't exist in Batch, but the Visual Studio task processor template has support for passing parameters via a resource file on the task. The resource file must contain a JSON map of key-value pairs, and must have the path ‘parameters.json’.

To pass parameters to a task, create a Dictionary<string, string> containing the task parameters, serialise it to JSON, upload it to storage, and pass a SAS URL to the blob and the path parameters.json as a resource file on the task, as shown here:

```
task.ResourceFiles = new [] {
    new ResourceFile(taskParametersSasUrl, "parameters.json"),
};
```

3.6 Update file reference code

In Batch Apps, the service managed input files in its internal storage, so the `TaskSpecifier.InputFiles` property needed only to specify file names on tasks. In Batch, the `CloudTask.ResourceFiles` property needs to specify not only the file names, but also the blob storage location from which to download each file, as shown here:

```
task.ResourceFiles = new [] {
    new ResourceFile(taskParametersSasUrl, "parameters.json"),
    new ResourceFile(blobSource1, filePath1),
    new ResourceFile(blobSource2, filePath2),
};
```

In addition, Batch does not provide a way to provide the job manager with a list of files for the job to process (you won’t usually want to use the `JobManagerTask.ResourceFiles` collection because that would cause the Batch service to download *all* job files to the job manager task). Depending on your scenario, you may be able to work out the files to pass to the tasks programmatically, for example by listing the contents of a storage container, or you may need to provide the list explicitly, for example as a JSON file passed to the job manager task as a resource file, as shown here:

```
var jobFilesListFile =
    Environment.ExpandEnvironmentVariables(@"%AZ_BATCH_TASK_WORKING_DIR%\JobFiles.json");
List<ResourceFile> jobFiles = ParseJsonToResourceFileList(jobFilesListFile);

// In this example, we create one CloudTask for each input file
int taskId = 0;
foreach (var jobFile in jobFiles)
{
    taskId++;
    yield return new CloudTask(taskId.ToString(), taskCommandLine)
    {
        ResourceFiles = new [] { jobFile },  // ignoring parameters.json in this example
    };
}
```

3.7 Add a merge task if needed

By default, the Batch Apps framework automatically appended a merge task to the sequence of tasks returned by the job splitter. If your job made use of this automatic merge task, your Batch code must explicitly create it. When creating it, keep in mind the following points:

- You need to invoke the merge code instead of the normal task processor code – see 2.9 Port your merge task code, for a discussion of this.

- The Batch Apps framework puts the default merge task in a stage after all other tasks. In Batch you will need to set up task dependencies to ensure that the merge task runs after all the tasks whose outputs it wants to merge.
- The Batch Apps framework automatically provided all previous task outputs as inputs to the merge task. You will need to specify these explicitly as resource files, or write code in your merge task processor to scan storage and download them from within the task processor.

3.8 Update any logging statements

In Batch Apps, you could log progress or diagnostic information to the job log using the `ILog` interface. Batch doesn't provide a built-in logging facility, but it does record standard output and error to files named stdout.txt and stderr.txt. Change your `Log` statements to `Console.Out.WriteLine` statements. You may want to change some statements to `Console.Error.WriteLine` instead, but this may make it harder for you to view a consolidated log, as some messages will then end up in stdout.txt and others in stderr.txt.

3.9 Save the logging files

In Batch Apps, logs were automatically saved and displayed in the Batch Apps preview portal. In Batch, you must save the stdout.txt and stderr.txt files to blob storage if you want them to be available after the compute node is torn down. Use the `TaskOutputStorage.SaveTrackedAsync` method to persist these files.

3.10 Make a note of the job manager program name

In Batch Apps, jobs were automatically associated with a job splitter through the job type. In Batch, jobs must be created with a reference to the job manager command line. (At least, that's the case if you're using the recommended Batch Apps migration architecture. In the general case, Batch jobs don't have to have a job manager; see the Batch documentation for more information.) You'll therefore need to know the name of the job manager program for when you port your client code. Check the assembly name in the Visual Studio project properties page and note it down for use in Step 5.

Step 4: Package and Deploy the Task Processor and Job Manager

4.1 Create an application package for the task programs

As discussed in the architecture overview, we will use the Batch application packages mechanism to deploy the job manager and task processor programs. We will deploy both programs using a single application package, though this is not required by the Batch service. This assumes that there are no conflicting dependencies between the two programs (for example, one depends on version 1.0 of a DLL and the other depends on version 2.0 of the same DLL). To accomplish this, do the following:

1. Create a working directory in which to assemble your task code package.
2. Build the job manager and task processor programs.
3. Open the job manager build directory.
4. Copy the job manager EXE and all its dependencies into your working directory.
5. Open the task processor build directory.

6. Copy the task processor EXE and all its dependencies into your working directory. You'll get some warnings about overwriting files (for example, the Batch and Storage DLLs) – make sure the files are identical to the ones they're overwriting.
7. Select all files in your working directory and send them to a compressed folder (or use your preferred zipping tool).

4.2 Deploy the application package containing the task programs

As with the application image, you must now install the task code package into your Batch account as an application package.

1. Open the Azure portal and navigate to your Batch account.
2. Click **Applications**.
3. Click **Add**.
4. Type the ID you chose in step 1.4, and version 1.0, and select the zip file you created in step 4.1.
5. Click **OK** to upload the task code package as an application package and make it available in your Batch account.

Step 5: Port the Client

5.1 Switch over to the Batch client

Your existing client has a reference to the Batch Apps client library. This must be replaced with a reference to the Batch client library, as follows:

1. Remove the Batch Apps NuGet package from your client project.
2. Add the Azure Batch NuGet package to your client project.
3. Change any "using Microsoft.Azure.Batch.Apps;" directives to "using Microsoft.Azure.Batch;".
4. Change `new BatchAppsClient()`to `await BatchClient.OpenAsync()`.
5. Change the service URL to be your Batch account URL instead of your Batch Apps service URL.
6. Change the account credentials to be a `BatchSharedKeyCredentials` instead of a `TokenCloudCredentials`.

5.2 Update job creation code

Job creation is slightly different in Batch from Batch Apps. You need to specify some additional information, and submit the job in a slightly different way:

- In Batch Apps, jobs were created using the `JobSubmission` class. In Batch, you use the `CloudJob` class. To create a job for submission, use `BatchClient.JobOperations.CreateJob`.
- In Batch Apps, the system assigned a GUID ID to each job. In Batch, you must assign the ID, and the ID is of string type. When porting your code to Batch Apps, we recommend using a GUID as follows:

```
job.Id = Guid.NewGuid().ToString();
```

- In Batch Apps, the system used the job type to locate the job splitter. In Batch, you must specify the command line of the job manager you created in step 3, and reference the task code application package so that Batch can find your program. You must also set the job manager's `KillJobOnCompletion` to false. You will also need to set up some environment variables required by the template job manager code; we will discuss those later. These changes are shown here:

```
job.JobManagerTask = new JobManagerTask {
  Id = Guid.NewGuid().ToString(),
  CommandLine = @"cmd /c %AZ_BATCH_APP_PACKAGE_{TaskPkgId}#1.0%\{jobmanager}.exe",
  KillJobOnCompletion = false,
  ApplicationPackageReferences = new [] {
    new ApplicationPackageReference { ApplicationId = "{TaskPkgId}", Version = "1.0" },
  },
};
```

Where *{TaskPkgId}* is the task code package ID you chose in step 1.4 and *{jobmanager}* is the program name you noted in step 3.10.

- Batch Apps provided job stage support for all jobs. In Batch, jobs must opt into task dependencies support. If your Batch Apps job uses stages, you must set the Batch job's `UsesTaskDependencies`, as shown here:

```
job.UsesTaskDependencies = true;
```

- In Batch Apps, jobs automatically complete when all tasks complete successfully, and jobs were terminated when any task fails. In Batch, if you want this behavior you must request it explicitly, as shown here:

```
job.OnAllTasksCompleted = OnAllTasksComplete.TerminateJob;
job.OnTaskFailure = OnTaskFailure.PerformExitOptionsJobAction;
```

- In Batch Apps, you specified parameters directly on the job object. In Batch, there is no built-in concept of job parameters; however, if you provide the job manager with a resource file named parameters.json then it will translate that into a parameters dictionary similar to the Batch Apps parameters collection. To do this, create a Dictionary<string, string> and populate it with your job parameters, serialise it to JSON, upload this to storage, and add it to the job manager task's resource files collection:

```
job.JobManagerTask = new JobManagerTask {
  Id = Guid.NewGuid().ToString(),
  CommandLine = @"cmd /c %AZ_BATCH_APP_PACKAGE_{TaskPkgId}#1.0%\{jobmanager}.exe",
  KillJobOnCompletion = false,
  ApplicationPackageReferences = new [] {
    new ApplicationPackageReference { ApplicationId = "{TaskPkgId}", Version = "1.0" },
  },
  ResourceFiles = new [] {
    new ResourceFile("https://{acct}.blob.core.windows.net/{ctnr}/parameter.json?{sas}",
     "parameters.json"),
  },
};
```

where *{acct}* is your storage account, *{ctnr}* the container where you uploaded the parameters file and *{sas}* the Shared Access Signature of the parameters file.

Finally, to submit the job to the Batch service, you call `CloudJob.CommitAsync`, as follows:

```
await job.CommitAsync();
```

5.3 Update file upload code

In Batch Apps, you could upload files directly to the Batch Apps service, and the service would manage the storage for you. In Batch, you have full control over storage. You need to upload your files to storage then pass a list of input files (and desired download locations) to each tasks. Because in this migration scenario tasks are created by a job manager, it means that you need to pass the list of input files to the job manager, which can then provide each task with its required files.

For each file you were previously uploading with your job:

- Upload the file to a blob in an Azure storage account. Do *not* use the linked storage account for this. The linked storage account must *only* be used by the Batch service and libraries.
- Create a Shared Access Signature (SAS) granting read permission for the blob. Ensure that the expiry time of the SAS allows plenty of time for the job to complete.
- Record the blob URL *including the SAS* together with the desired file location on compute nodes.

To pass the list of files to the job manager, so it can assign them as resource files to the various tasks, do the following:

- Write the list of URL-file location pairs to a string, e.g. in JSON or entry-per-line format.
- Upload this string to a blob in an Azure storage account (again, do not use the linked storage account). You can use the `CloudBlockBlob.UploadTextAsync` method to do this.
- Create a SAS granting read permission to the blob.
- Add the SAS URL and file location to the job manager task's `ResourceFiles` collection.

```
These steps are shown here:job.JobManagerTask = new JobManagerTask {
  Id = Guid.NewGuid().ToString(),
  CommandLine = @"cmd /c %AZ_BATCH_APP_PACKAGE_{TaskPkgId}#1.0%\{jobmanager}.exe",
  KillJobOnCompletion = false,
  ApplicationPackageReferences = new [] {
    new ApplicationPackageReference { ApplicationId = "{TaskPkgId}", Version = "1.0" },
  },
  ResourceFiles = new [] {
    new ResourceFile("https://{acct}.blob.core.windows.net/{ctnr}/parameter.json?{sas}",
     "parameters.json"),
    new ResourceFile("https://{acct}.blob.core.windows.net/{ctnr}/JobFiles.json?{sas}",
      "JobFiles.json"),
  },
};
```

where *{acct}* is your storage account, *{ctnr}* the container where you uploaded the files list file and *{sas}* the Shared Access Signature of the files list file.

5.4 Set up storage for the job

In Batch Apps, when a task processor returned a list of files to persist, the framework took care of uploading them to Azure blob storage. In Batch, the task processor needs to persist files directly from its own code. The task processor therefore needs your storage account credentials, which it expects to receive via environment variables. The easiest way to make the storage account available to all tasks is to set up these environment variables at the job level, using the `CloudJob.CommonEnvironmentSettings` property, as shown here:

```
job.CommonEnvironmentSettings = new [] {
  new EnvironmentSetting("LINKED_STORAGE_ACCOUNT", "{storageAccountName}"),
  new EnvironmentSetting("LINKED_STORAGE_KEY", "{storageAccountKey}"),
};
```

where *{storageAccountName}* is the name of the storage account that you linked to the Batch account in step 1.1 and *{storageAccountKey}* is the key of that storage account (which you can copy from the Azure portal).

5.5 Provide Batch credentials for the job manager

In Batch Apps, when the job splitter returned tasks, the framework took care of adding them to the job. In Batch, the job manager does this using the Batch API. You don't need to write any code yourself to do this; the necessary code was generated when you created your job manager project from the Visual Studio template. The job manager therefore needs your Batch account credentials, which it expects to receive via environment variables. Your client must set these up using the job manager task's `EnvironmentSettings` property, as shown here:

```
job.JobManagerTask = new JobManagerTask {
  Id = Guid.NewGuid().ToString(),
  CommandLine = @"cmd /c %AZ_BATCH_APP_PACKAGE_{TaskPkgId}#1.0%\{jobmanager}.exe",
  KillJobOnCompletion = false,
  ApplicationPackageReferences = new [] {
    new ApplicationPackageReference { ApplicationId = "{TaskPkgId}", Version = "1.0" },
  },
  ResourceFiles = new [] {
    new ResourceFile("https://{acct}.blob.core.windows.net/{ctnr}/parameter.json?{sas}",
     "parameters.json"),
    new ResourceFile("https://{acct}.blob.core.windows.net/{ctnr}/files.txt?{sas}", "files.txt"),
  },
  EnvironmentSettings = new [] {
    new EnvironmentSetting("YOUR_BATCH_URL", "{batchAccountUrl}"),
    new EnvironmentSetting("YOUR_BATCH_KEY", "{batchAccountKey}"),
  }
};
```

where *{batchAccountUrl}* is the URL of your Batch account, which you can copy from the Azure portal (it will be of the form https://*account_name*.*region*.batch.azure.com), and *{batchAccountKey}* is the key of your Batch account in base64-encoded form, which you can also copy from the portal.

5.6 Change how you detect job outcome

Batch Apps marked jobs and tasks as 'completed' or 'failed.' Batch doesn't judge success or failure, only whether the job or task has terminated. This means you need to change the way you detect outcomes.

To detect if a task has succeeded or failed, you can do the following:

- Call `CloudJob.GetTaskAsync()` or `CloudJob.ListTasks()`.
- Examine the `CloudTask.State`.
- If `State` is `Completed`, examine the `CloudTask.ExitCode`.
- An exit code of 0 represents success. Any other exit code represents failure.

To detect if a job has succeeded or failed, if you are using the built-in completion features, you can do the following:

- Call `CloudJob.Refresh()` or `JobOperations.GetJobAsync()`.
- Examine the `CloudJob.State`.

- If `State` is `Completed`, examine the `CloudJob.ExecutionInformation.TerminateReason`.
- If `TerminateReason` is `AllTasksComplete`, then all tasks completed successfully; that is, the job succeeded in the way they did in Batch Apps. If `TerminateReason` is `TaskFailed`, then the job was terminated due to a task failing. Other reason strings can reflect other reasons the job terminated, such as a client program calling the Terminate Job API; you shouldn't see these during a Batch Apps migration.

Note
If you want to be able to re-process tasks, you should have configured your tasks to disable the job on failure, rather than terminate it. In this case, if a task fails, the job state will be `Disabled`, and there will no `TerminateReason`. You will need to add additional checks for this case, and should have a procedure for terminating a job if you decide you do not want to re-process it (you can terminate a job through the portal).

5.7 Change how you retrieve job and task outputs

In Batch Apps, you use the Batch Apps Client to retrieve job and task outputs. In Batch, you retrieve these directly from storage.

Use the `TaskOutputStorage.ListOutputs` and `GetOutputAsync` methods to list and retrieve task outputs, and `JobOutputStorage.ListOutputs` and `GetOutputAsync` to list and retrieve job outputs.

Step 6: Test Your Solution

To run test jobs and diagnose problems in Batch, you can use the Azure portal. The Batch samples repository on GitHub (https://github.com/Azure/azure-batch-samples/) also includes Batch Explorer which is a tool that you can customise to your own diagnostic and testing requirements.

You can also test your job manager and task processor locally, for example by running them under a debugger. You will need to set up appropriate environment variables. You can also test the job splitter and task processor classes in isolation by faking the `IConfiguration` interface.

Both the job manager and task processor trap all exceptions in order to provide good pointers to the location of failures when running in the production environment. When testing, you can use this diagnostic information, or you can turn on first-chance exceptions to catch exceptions at the point they are thrown.

Note that some categories of error will result in your programs not being run at all in Batch. In this case the Batch service should provide diagnostics, for example via a 'scheduling error' or an error detail API response. Likely causes for these errors are:

- Incorrect or missing resource files. Be sure that all resource files have valid SAS tokens and that the blob storage locations are correct. Remember that Batch does not manage storage for you in the way that Batch Apps did.
- Incorrect application package reference. Be sure that your package IDs match the ones used during deployment. Remember that Batch does not automatically map tasks to cloud assemblies or application images in the way that Batch Apps did.

- Incorrect task command line. This could occur if you misspell an application package location environment variable (AZ_BATCH_APP_PACKAGE_...) or program name when creating a task.

Getting Help

If you need additional information or advice when migrating your Batch Apps solution to Batch, or if you are having trouble diagnosing or fixing a problem with your ported code, don't hesitate to reach out to the Batch team by using the following MSDN forum:

https://social.msdn.microsoft.com/forums/azure/home?forum=azurebatch