# Job Manager and Task Processor Templates for Azure Batch

## 1 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. With Batch, you can schedule compute-intensive work to run on a managed collection of virtual machines, and have Batch automatically scale compute resources to meet the needs of your jobs.

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 document discusses only information applicable to these two templates, and assumes that you are familiar with the Batch service and key concepts related to it, like 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, Get started with the Azure Batch library for .NET, and Batch feature overview for developers, in the Batch documentation.

## 2 High-level overview

The Job Manager and Task Processor templates can be used to create two useful components:

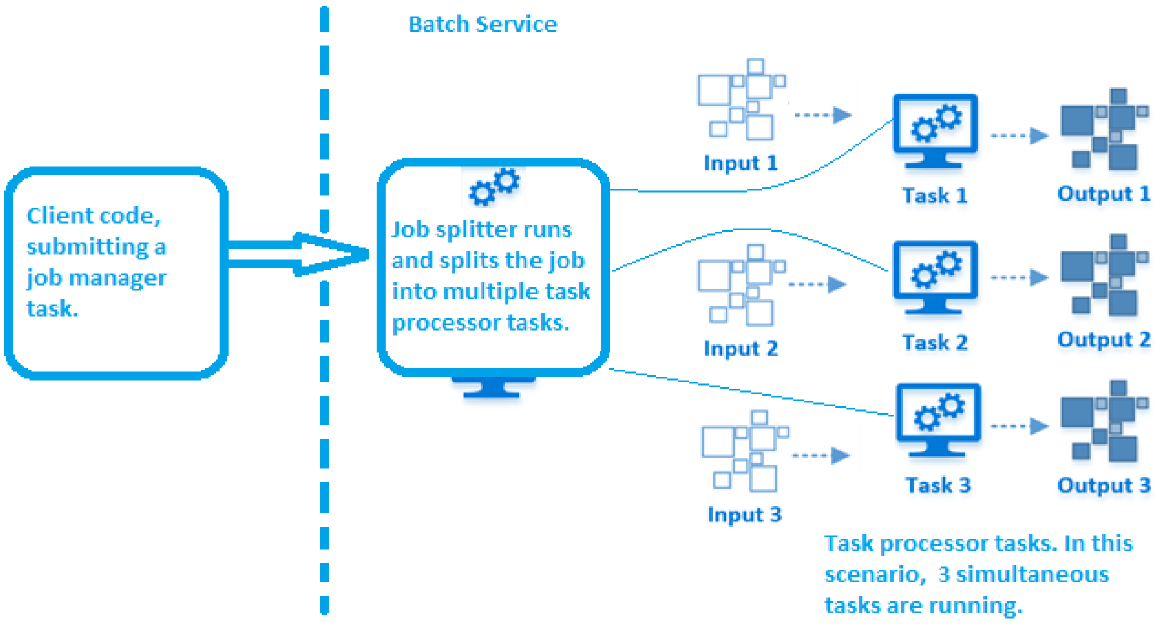
- A job manager task that implements a job splitter that can break a job down into multiple tasks that can run independently, in parallel.
- A task processor that can be used to efficiently invoke the application executables and related actions for a given task.

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 manager task to the Batch service on Azure.
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.


Batch Service
Client code, submitting a job manager task.
Job splitter runs and splits the job into multiple task processor tasks.
Input 1
Task 1
Output 1
Input 2
Task 2
Output 2
Input 3
Task 3
Output 3
Task processor tasks. In this scenario, 3 simultaneous tasks are running.

## 3 Before You Begin

### 3.1 Prerequisites

To use the Batch templates, you will need the following:

- A computer with Visual Studio 2015, or newer, already installed on it.
- 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 https://msdn.microsoft.com/library/dd293638.aspx). 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: https://go.microsoft.com/fwlink/?linkid=820714.

- If you plan to use the Application Packages feature to deploy the job manager and task processor to the Batch compute nodes, you will need to link a storage account to your Batch account.

### 3.2 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 and create duplicates of your entire job manager and task processor 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**.

## 4 About the Job Manager Template

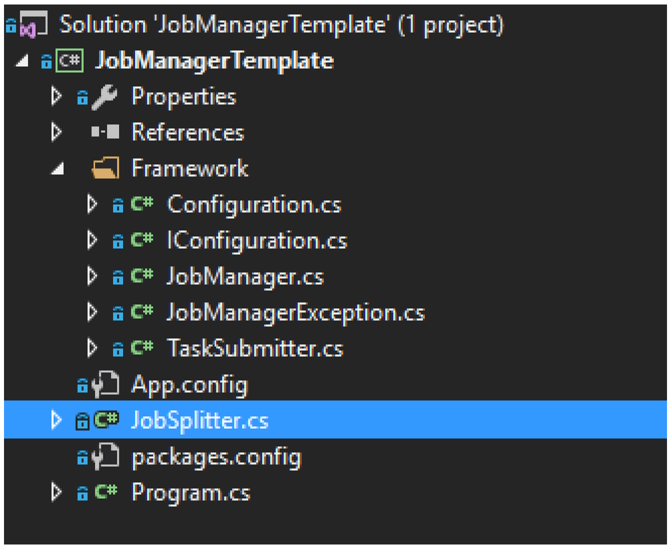
The Job Manager template helps you to implement a job manager task that can perform the following actions:

- 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, in the Batch documentation.

### 4.1 Job Manager template files and their purpose

The files responsible for the background work done by the template code reside inside the Framework folder. You should not normally need to touch these files unless you want to customize the low-level workings of the job manager. For your application logic, you should work within the JobSplitter.cs file, plus any files that you create to support your job splitter implementation. This section describes the different files and their code structure, and explains what each class does.



The different files in the Job Manager template are:

- **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.
- **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.
- **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.

## 4.2 How to use the Job Manager Template

To add a job manager to the solution that you created earlier, follow these steps:

1. Open your existing solution in Visual Studio 2015.
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 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.

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 show 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 first familiarize yourself with how Batch works by reviewing the Batch documentation and trying out a few of the Batch code samples.

## 4.3 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 `JobManagerException` 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.

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.

# 5 About the Task Processor Template

A Task Processor template helps you to implement a task processor that can perform the following actions:

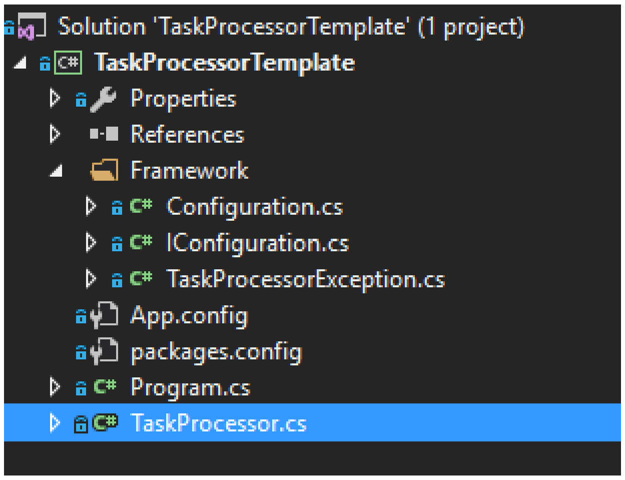
- Set up the information required by each Batch task to run.
- Run all actions required by each Batch task.
- Save task outputs to persistent storage.

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.

## 5.1 Task Processor template files and their purpose

The files responsible for the background work done by the template code reside inside the Framework folder. You should not normally need to touch these files unless you want to customize the low-level workings of the task processor. For your application logic, you should work within the TaskProcessor.cs file, plus any files that you create to support your task processor implementation This section describes the different files and their code structure, and explains what each class does.



The different files in the Task Processor template are:

- **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.
- **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
- **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.

## 5.2 How to use the Task Processor template

To add a task processor to the solution that you created earlier, follow these steps:

1. Open your existing solution in Visual Studio 2015.

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.

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.

## 5.3 Implementing and running the Run() method

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.

## 5.4 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 `TaskProcessorException` 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.

# 6 How to pass parameters and environment variables from the client code

## 6.1 Passing 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.

## 6.2 Passing 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.

## 6.3 Passing 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.