Here is the code that implements the orchestrator function:

**C#**

#r "Microsoft.Azure.WebJobs.Extensions.DurableTask"

public static async Task<long> Run(DurableOrchestrationContext backupCon-

text)

{

string rootDirectory = Environment.ExpandEnvironmentVariables(backup-Context.GetInput<string>() ?? "");

|  |
| --- |
|  |

|  |
| --- |
|  |

**116** Implement Azure functions

if (string.IsNullOrEmpty(rootDirectory))

{

throw new ArgumentException("A directory path is required as an input.");

}

if (!Directory.Exists(rootDirectory))

{

throw new DirectoryNotFoundException($"Could not find a directory named '{rootDirectory}'.");

}

string[] files = await backupContext.CallActivityAsync<string[]>( "E2\_GetFileList",

rootDirectory);

var tasks = new Task<long>[files.Length];

for (int i = 0; i < files.Length; i++)

{

tasks[i] = backupContext.CallActivityAsync<long>( "E2\_CopyFileToBlob",

files[i]);

}

await Task.WhenAll(tasks);

long totalBytes = tasks.Sum(t => t.Result);

return totalBytes;

}

**JavaScript (Functions v2 only)**

const df = require("durable-functions");

module.exports = df.orchestrator(function\*(context){

const rootDirectory = context.df.getInput(); if (!rootDirectory) {

throw new Error("A directory path is required as an input.");

}

const files = yield context.df.callActivity("E2\_GetFileList", rootDirec-tory);

* Backup Files and save Promises into array const tasks = [];

for (const file of files) { tasks.push(context.df.callActivity("E2\_CopyFileToBlob", file));

}

* wait for all the Backup Files Activities to complete, sum total

Implement Durable Functions **117**

bytes

const results = yield context.df.Task.all(tasks);

const totalBytes = results.reduce((prev, curr) => prev + curr, 0);

* return results; return totalBytes;

});

This orchestrator function essentially does the following:

1. Takes a rootDirectory value as an input parameter.
2. Calls a function to get a recursive list of files under rootDirectory.
3. Makes multiple parallel function calls to upload each file into Azure Blob Storage.
4. Waits for all uploads to complete.
5. Returns the sum total bytes that were uploaded to Azure Blob Storage.

Notice the await Task.WhenAll(tasks); (C#) and yield context.df.Task.all(tasks); (JS) line. All the calls to the E2\_CopyFileToBlob function were *not* awaited. This is intentional to allow them to run in parallel. When we pass this array of tasks to Task.WhenAll, we get back a task that won't complete *until all the copy operations have completed*. If you're familiar with the Task Parallel Library (TPL) in .NET, then this is not new to you. The difference is that these tasks could be running on multiple VMs concurrently, and the Durable Functions extension ensures that the end-to-end execution is resilient to process recycling.

Tasks are very similar to the JavaScript concept of promises. However, Promise.all has some differenc-es from Task.WhenAll. The concept of Task.WhenAll has been ported over as part of the dura-ble-functions JavaScript module and is exclusive to it.

After awaiting from Task.WhenAll (or yielding from context.df.Task.all), we know that all function calls have completed and have returned values back to us. Each call to E2\_CopyFileToBlob returns the number of bytes uploaded, so calculating the sum total byte count is a matter of adding all those return values together.

**Helper activity functions**

The helper activity functions, as with other samples, are just regular functions that use the activi-tyTrigger trigger binding. For example, the *function.json* file for E2\_GetFileList looks like the following:

{

"bindings": [

{

"name": "rootDirectory",

"type": "activityTrigger",

"direction": "in"

}

],

"disabled": false

}

|  |
| --- |
|  |

And here is the implementation:

|  |
| --- |
|  |

**118** Implement Azure functions

**C#**

#r "Microsoft.Azure.WebJobs.Extensions.DurableTask"

#r "Microsoft.Extensions.Logging"

public static string[] Run(string rootDirectory, ILogger log)

{

string[] files = Directory.GetFiles(rootDirectory, "\*", SearchOption. AllDirectories);

log.LogInformation($"Found {files.Length} file(s) under {rootDirecto-ry}.");

return files;

}

**JavaScript (Functions v2 only)**

const readdirp = require("readdirp");

module.exports = function (context, rootDirectory) { context.log(`Searching for files under '${rootDirectory}'...`); const allFilePaths = [];

readdirp(

{root: rootDirectory, entryType: 'all'},

function (fileInfo) {

if (!fileInfo.stat.isDirectory()) {

allFilePaths.push(fileInfo.fullPath);

}

},

function (err, res) {

if (err) {

throw err;

}

context.log(`Found ${allFilePaths.length} under ${rootDirecto-

ry}.`);

context.done(null, allFilePaths);

}

);

};

The JavaScript implementation of E2\_GetFileList uses the readdirp module to recursively read the directory structure.

**Note:** You might be wondering why you couldn't just put this code directly into the orchestrator function.You could, but this would break one of the fundamental rules of orchestrator functions, which is that they should never do I/O, including local file system access.

The function.json file for E2\_CopyFileToBlob is similarly simple:

Implement Durable Functions **119**

{

"bindings": [

{

"name": "filePath",

"type": "activityTrigger",

"direction": "in"

}

],

"disabled": false

}

The C# implementation is also pretty straightforward. It happens to use some advanced features of Azure Functions bindings (that is, the use of the Binder parameter), but you don't need to worry about those details for the purpose of this walkthrough.

**C#**

#r "Microsoft.Azure.WebJobs.Extensions.DurableTask"

#r "Microsoft.Azure.WebJobs.Extensions.Storage"

#r "Microsoft.Extensions.Logging"

#r "Microsoft.WindowsAzure.Storage"

using Microsoft.WindowsAzure.Storage.Blob;

public static async Task<long> Run(

string filePath,

Binder binder,

ILogger log)

{

long byteCount = new FileInfo(filePath).Length;

* strip the drive letter prefix and convert to forward slashes string blobPath = filePath

.Substring(Path.GetPathRoot(filePath).Length)

.Replace('\\', '/');

string outputLocation = $"backups/{blobPath}";

log.LogInformation($"Copying '{filePath}' to '{outputLocation}'. Total bytes = {byteCount}.");

// copy the file contents into a blob

using (Stream source = File.Open(filePath, FileMode.Open, FileAccess. Read, FileShare.ReadWrite))

using (Stream destination = await binder.BindAsync<CloudBlobStream>( new BlobAttribute(outputLocation)))

{

await source.CopyToAsync(destination);

}

return byteCount;

|  |
| --- |
|  |

|  |
| --- |
|  |

**120** Implement Azure functions

}

**JavaScript (Functions v2 only)**

The JavaScript implementation does not have access to the Binder feature of Azure Functions, so the Azure Storage SDK for Node takes its place.

const fs = require("fs");

const path = require("path");

const storage = require("azure-storage");

module.exports = function (context, filePath) { const container = "backups";

const root = path.parse(filePath).root;

const blobPath = filePath

.substring(root.length)

.replace("\\", "/");

const outputLocation = `backups/${blobPath}`;

const blobService = storage.createBlobService(process.env['AzureWeb-JobsStorage']);

blobService.createContainerIfNotExists(container, (error) => { if (error) {

throw error;

}

fs.stat(filePath, function (error, stats) {

if (error) {

throw error;

}

context.log(`Copying '${filePath}' to '${outputLocation}'. Total bytes = ${stats.size}.`);

const readStream = fs.createReadStream(filePath);

blobService.createBlockBlobFromStream(container, blobPath, readStream, stats.size, function (error) {

if (error) {

throw error;

}

context.done(null, stats.size);

});

});

});

};

The implementation loads the file from disk and asynchronously streams the contents into a blob of the same name in the “backups” container. The return value is the number of bytes copied to storage, that is then used by the orchestrator function to compute the aggregate sum.

Implement Durable Functions **121**

**Note:** This is a perfect example of moving I/O operations into anactivityTriggerfunction. Not onlycan the work be distributed across many different VMs, but you also get the benefits of checkpointing the progress. If the host process gets terminated for any reason, you know which uploads have already completed.

**Run the sample**

You can start the orchestration by sending the following HTTP POST request.

POST http://{host}/orchestrators/E2\_BackupSiteContent

Content-Type: application/json

Content-Length: 20

"D:\\home\\LogFiles"

Note: The HttpStart function that you are invoking only works with JSON-formatted content. For this reason, the Content-Type: application/json header is required and the directory path is encod-ed as a JSON string. Moreover, HTTP snippet assumes there is an entry in the host.json file which removes the default api/ prefix from all HTTP trigger functions URLs. You can find the markup for this configuration in the host.json file in the samples.

This HTTP request triggers the E2\_BackupSiteContent orchestrator and passes the string D:\home\ LogFiles as a parameter. The response provides a link to get the status of the backup operation:

HTTP/1.1 202 Accepted

Content-Length: 719

Content-Type: application/json; charset=utf-8

Location: http://{host}/admin/extensions/DurableTaskExtension/instances/ b4e9bdcc435d460f8dc008115ff0a8a9?taskHub=DurableFunction-sHub&connection=Storage&code={systemKey}

(...trimmed...)

Depending on how many log files you have in your function app, this operation could take several minutes to complete. You can get the latest status by querying the URL in the Location header of the previous HTTP 202 response.

GET http://{host}/admin/extensions/DurableTaskExtension/instances/b4e9bdc-c435d460f8dc008115ff0a8a9?taskHub=DurableFunctionsHub&con-nection=Storage&code={systemKey}

HTTP/1.1 202 Accepted

Content-Length: 148

Content-Type: application/json; charset=utf-8

Location: http://{host}/admin/extensions/DurableTaskExtension/instances/ b4e9bdcc435d460f8dc008115ff0a8a9?taskHub=DurableFunction-sHub&connection=Storage&code={systemKey}

{"runtimeStatus":"Running","input":"D:\\home\\LogFiles","output":null,"cre-atedTime":"2017-06-29T18:50:55Z","lastUpdatedTime":"2017-06-29T18:51:16Z"}

|  |
| --- |
|  |

|  |
| --- |
|  |

**122** Implement Azure functions

In this case, the function is still running. You are able to see the input that was saved into the orchestrator state and the last updated time. You can continue to use the Location header values to poll for comple-tion. When the status is “Completed”, you see an HTTP response value similar to the following:

HTTP/1.1 200 OK

Content-Length: 152

Content-Type: application/json; charset=utf-8

{"runtimeStatus":"Completed","input":"D:\\home\\LogFiles","out-put":452071,"createdTime":"2017-06-29T18:50:55Z","lastUpdated-Time":"2017-06-29T18:51:26Z"}

Now you can see that the orchestration is complete and approximately how much time it took to com-plete. You also see a value for the output field, which indicates that around 450 KB of logs were upload-ed.

**Visual Studio sample code**

Here is the orchestration as a single C# file in a Visual Studio project:

* Copyright (c) .NET Foundation. All rights reserved.
* Licensed under the MIT License. See LICENSE in the project root for license information.

using System.IO;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.Azure.WebJobs;

using Microsoft.Extensions.Logging;

using Microsoft.WindowsAzure.Storage.Blob;

namespace VSSample

{

public static class BackupSiteContent

{

[FunctionName("E2\_BackupSiteContent")]

public static async Task<long> Run(

[OrchestrationTrigger] DurableOrchestrationContext backupCon-

text)

{

string rootDirectory = backupContext.GetInput<string>()?.

Trim();

if (string.IsNullOrEmpty(rootDirectory))

{

rootDirectory = Directory.GetParent(typeof(BackupSiteCon-tent).Assembly.Location).FullName;

}

string[] files = await backupContext.CallActivityAsyn-

c<string[]>(

"E2\_GetFileList",

rootDirectory);

Implement Durable Functions **123**

var tasks = new Task<long>[files.Length];

for (int i = 0; i < files.Length; i++)

{

tasks[i] = backupContext.CallActivityAsync<long>( "E2\_CopyFileToBlob",

files[i]);

}

await Task.WhenAll(tasks);

long totalBytes = tasks.Sum(t => t.Result);

return totalBytes;

}

[FunctionName("E2\_GetFileList")]

public static string[] GetFileList(

[ActivityTrigger] string rootDirectory,

ILogger log)

{

log.LogInformation($"Searching for files under '{rootDirecto-

ry}'...");

string[] files = Directory.GetFiles(rootDirectory, "\*", SearchOption.AllDirectories);

log.LogInformation($"Found {files.Length} file(s) under {rootDi-

rectory}.");

return files;

}

[FunctionName("E2\_CopyFileToBlob")]

public static async Task<long> CopyFileToBlob( [ActivityTrigger] string filePath,

Binder binder,

ILogger log)

{

long byteCount = new FileInfo(filePath).Length;

* strip the drive letter prefix and convert to forward slashes string blobPath = filePath

.Substring(Path.GetPathRoot(filePath).Length)

.Replace('\\', '/');

string outputLocation = $"backups/{blobPath}";

log.LogInformation($"Copying '{filePath}' to '{outputLocation}'. Total bytes = {byteCount}.");

// copy the file contents into a blob

using (Stream source = File.Open(filePath, FileMode.Open, File-Access.Read, FileShare.Read))

using (Stream destination = await binder.BindAsync<CloudBlob-

|  |
| --- |
|  |

|  |
| --- |
|  |