

Contents

Durable Functions Documentation

Overview

[About Azure Functions](#)

[About Durable Functions](#)

[Serverless comparison](#)

[Durable Functions versions](#)

Quickstarts

[Create durable function - C#](#)

[Create durable function - JavaScript](#)

[Create durable function - Python](#)

[Create durable function - PowerShell](#)

[Create durable function - Java](#)

Tutorials

[Function chaining](#)

[Fan-out/fan-in](#)

[Monitors \(C#, JavaScript\)](#)

[Monitors \(Python\)](#)

[Human interaction](#)

[Publish to Event Grid](#)

Samples

[Code samples](#)

[Azure CLI](#)

Concepts

[Function types](#)

[Orchestrations](#)

[Orchestrator code constraints](#)

[Sub-orchestrations](#)

[Custom orchestration status](#)

[Timers](#)

- [External events](#)
- [Error handling](#)
- [Eternal orchestrations](#)
- [Singleton orchestrations](#)
- [HTTP features](#)
- [Stateful entities](#)
- [Task hubs](#)
- [Instance management](#)
- [Versioning](#)
- [Storage providers](#)
- [Performance and scale](#)
- [High availability](#)
- [Data persistence and serialization](#)
- [Reference](#)
 - [Host.json settings](#)
 - [Bindings](#)
 - [HTTP API](#)
 - [.NET API](#)
 - [Node.js API](#)
 - [Python API](#)
 - [Azure storage provider](#)
- [How-to guides](#)
 - [Develop](#)
 - [Create in Azure portal](#)
 - [Manage connections](#)
 - [Unit testing](#)
 - [Create as WebJobs](#)
 - [Durable entities in .NET](#)
 - [Deploy](#)
 - [Continuous deployment](#)
 - [Zip deployment](#)
 - [Run from package](#)

[Zero downtime deployment](#)

[Monitor](#)

[Diagnostics](#)

[Monitoring](#)

[Pricing](#)

[Resources](#)

[Build your skills with Microsoft Learn training](#)

[Azure Roadmap](#)

[Pricing](#)

[Pricing calculator](#)

[Regional availability](#)

[Videos](#)

[Microsoft Q&A question page](#)

[Stack Overflow](#)

[Twitter](#)

[Provide product feedback](#)

[Durable Functions GitHub repository](#)

[Service updates](#)

Introduction to Azure Functions

10/5/2022 • 2 minutes to read • [Edit Online](#)

Azure Functions is a serverless solution that allows you to write less code, maintain less infrastructure, and save on costs. Instead of worrying about deploying and maintaining servers, the cloud infrastructure provides all the up-to-date resources needed to keep your applications running.

You focus on the pieces of code that matter most to you, and Azure Functions handles the rest.

We often build systems to react to a series of critical events. Whether you're building a web API, responding to database changes, processing IoT data streams, or even managing message queues - every application needs a way to run some code as these events occur.

To meet this need, Azure Functions provides "compute on-demand" in two significant ways.

First, Azure Functions allows you to implement your system's logic into readily available blocks of code. These code blocks are called "functions". Different functions can run anytime you need to respond to critical events.

Second, as requests increase, Azure Functions meets the demand with as many resources and function instances as necessary - but only while needed. As requests fall, any extra resources and application instances drop off automatically.

Where do all the compute resources come from? Azure Functions [provides as many or as few compute resources as needed](#) to meet your application's demand.

Providing compute resources on-demand is the essence of [serverless computing](#) in Azure Functions.

Scenarios

In many cases, a function [integrates with an array of cloud services](#) to provide feature-rich implementations.

The following are a common, *but by no means exhaustive*, set of scenarios for Azure Functions.

IF YOU WANT TO...	THEN...
Build a web API	Implement an endpoint for your web applications using the HTTP trigger
Process file uploads	Run code when a file is uploaded or changed in blob storage
Build a serverless workflow	Chain a series of functions together using durable functions
Respond to database changes	Run custom logic when a document is created or updated in Cosmos DB
Run scheduled tasks	Execute code on pre-defined timed intervals
Create reliable message queue systems	Process message queues using Queue Storage , Service Bus , or Event Hubs

IF YOU WANT TO...	THEN...
Analyze IoT data streams	Collect and process data from IoT devices
Process data in real time	Use Functions and SignalR to respond to data in the moment

As you build your functions, you have the following options and resources available:

- **Use your preferred language:** Write functions in [C#, Java, JavaScript, PowerShell, or Python](#), or use a [custom handler](#) to use virtually any other language.
- **Automate deployment:** From a tools-based approach to using external pipelines, there's a [myriad of deployment options](#) available.
- **Troubleshoot a function:** Use [monitoring tools](#) and [testing strategies](#) to gain insights into your apps.
- **Flexible pricing options:** With the [Consumption](#) plan, you only pay while your functions are running, while the [Premium](#) and [App Service](#) plans offer features for specialized needs.

Next Steps

[Get started through lessons, samples, and interactive tutorials](#)

What are Durable Functions?

10/5/2022 • 22 minutes to read • [Edit Online](#)

Durable Functions is an extension of [Azure Functions](#) that lets you write stateful functions in a serverless compute environment. The extension lets you define stateful workflows by writing *orchestrator functions* and stateful entities by writing *entity functions* using the Azure Functions programming model. Behind the scenes, the extension manages state, checkpoints, and restarts for you, allowing you to focus on your business logic.

Supported languages

Durable Functions is designed to work with all Azure Functions programming languages but may have different minimum requirements for each language. The following table shows the minimum supported app configurations:

LANGUAGE STACK	AZURE FUNCTIONS RUNTIME VERSIONS	LANGUAGE WORKER VERSION	MINIMUM BUNDLES VERSION
.NET / C# / F#	Functions 1.0+	In-process (GA) Out-of-process (preview)	n/a
JavaScript/TypeScript	Functions 2.0+	Node 8+	2.x bundles
Python	Functions 2.0+	Python 3.7+	2.x bundles
PowerShell	Functions 3.0+	PowerShell 7+	2.x bundles
Java (preview)	Functions 3.0+	Java 8+	4.x bundles

Like Azure Functions, there are templates to help you develop Durable Functions using [Visual Studio 2019](#), [Visual Studio Code](#), and the [Azure portal](#).

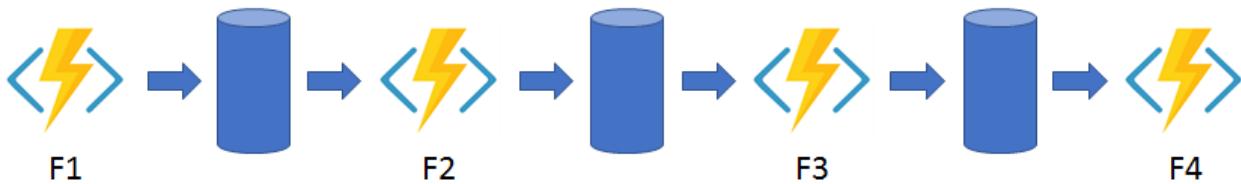
Application patterns

The primary use case for Durable Functions is simplifying complex, stateful coordination requirements in serverless applications. The following sections describe typical application patterns that can benefit from Durable Functions:

- [Function chaining](#)
- [Fan-out/fan-in](#)
- [Async HTTP APIs](#)
- [Monitoring](#)
- [Human interaction](#)
- [Aggregator \(stateful entities\)](#)

Pattern #1: Function chaining

In the function chaining pattern, a sequence of functions executes in a specific order. In this pattern, the output of one function is applied to the input of another function.



You can use Durable Functions to implement the function chaining pattern concisely as shown in the following example.

In this example, the values `F1`, `F2`, `F3`, and `F4` are the names of other functions in the same function app. You can implement control flow by using normal imperative coding constructs. Code executes from the top down. The code can involve existing language control flow semantics, like conditionals and loops. You can include error handling logic in `try / catch / finally` blocks.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```

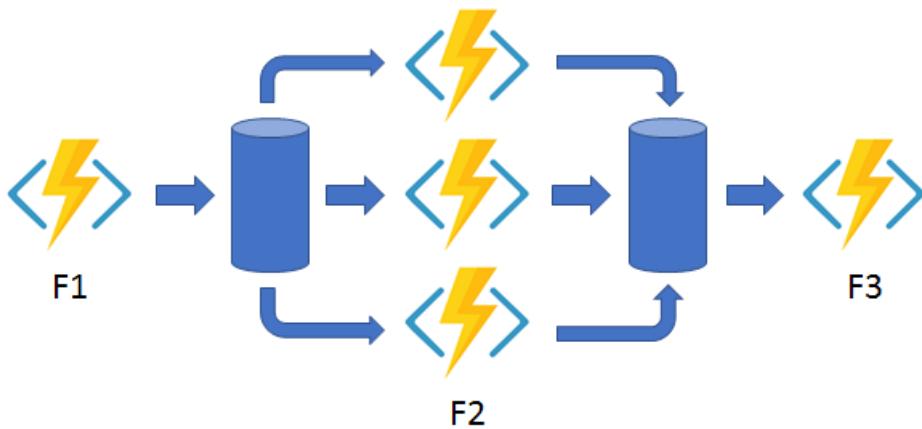
[FunctionName("Chaining")]
public static async Task<object> Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    try
    {
        var x = await context.CallActivityAsync<object>("F1", null);
        var y = await context.CallActivityAsync<object>("F2", x);
        var z = await context.CallActivityAsync<object>("F3", y);
        return await context.CallActivityAsync<object>("F4", z);
    }
    catch (Exception)
    {
        // Error handling or compensation goes here.
    }
}

```

You can use the `context` parameter to invoke other functions by name, pass parameters, and return function output. Each time the code calls `await`, the Durable Functions framework checkpoints the progress of the current function instance. If the process or virtual machine recycles midway through the execution, the function instance resumes from the preceding `await` call. For more information, see the next section, Pattern #2: Fan out/fan in.

Pattern #2: Fan out/fan in

In the fan out/fan in pattern, you execute multiple functions in parallel and then wait for all functions to finish. Often, some aggregation work is done on the results that are returned from the functions.



With normal functions, you can fan out by having the function send multiple messages to a queue. Fanning back in is much more challenging. To fan in, in a normal function, you write code to track when the queue-triggered functions end, and then store function outputs.

The Durable Functions extension handles this pattern with relatively simple code:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```

[FunctionName("FanOutFanIn")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    var parallelTasks = new List<Task<int>>();

    // Get a list of N work items to process in parallel.
    object[] workBatch = await context.CallActivityAsync<object[]>("F1", null);
    for (int i = 0; i < workBatch.Length; i++)
    {
        Task<int> task = context.CallActivityAsync<int>("F2", workBatch[i]);
        parallelTasks.Add(task);
    }

    await Task.WhenAll(parallelTasks);

    // Aggregate all N outputs and send the result to F3.
    int sum = parallelTasks.Sum(t => t.Result);
    await context.CallActivityAsync("F3", sum);
}

```

The fan-out work is distributed to multiple instances of the `F2` function. The work is tracked by using a dynamic list of tasks. `Task.WhenAll` is called to wait for all the called functions to finish. Then, the `F2` function outputs are aggregated from the dynamic task list and passed to the `F3` function.

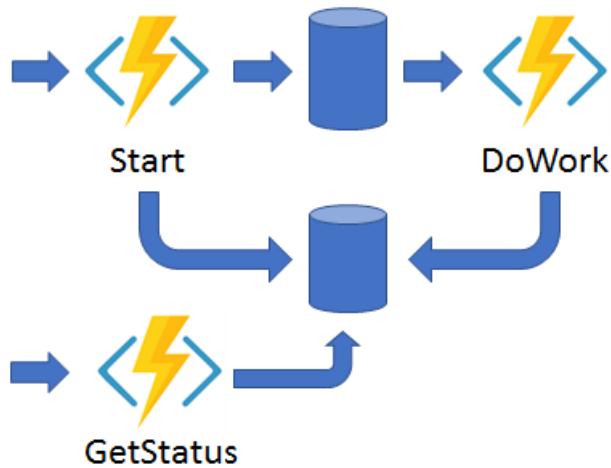
The automatic checkpointing that happens at the `await` call on `Task.WhenAll` ensures that a potential midway crash or reboot doesn't require restarting an already completed task.

NOTE

In rare circumstances, it's possible that a crash could happen in the window after an activity function completes but before its completion is saved into the orchestration history. If this happens, the activity function would re-run from the beginning after the process recovers.

Pattern #3: Async HTTP APIs

The async HTTP API pattern addresses the problem of coordinating the state of long-running operations with external clients. A common way to implement this pattern is by having an HTTP endpoint trigger the long-running action. Then, redirect the client to a status endpoint that the client polls to learn when the operation is finished.



Durable Functions provides **built-in support** for this pattern, simplifying or even removing the code you need to write to interact with long-running function executions. For example, the Durable Functions quickstart samples ([C#](#), [JavaScript](#), [Python](#), [PowerShell](#), and [Java](#)) show a simple REST command that you can use to start new orchestrator function instances. After an instance starts, the extension exposes webhook HTTP APIs that query the orchestrator function status.

The following example shows REST commands that start an orchestrator and query its status. For clarity, some protocol details are omitted from the example.

```

> curl -X POST https://myfunc.azurewebsites.net/api/orchestrators/DoWork -H "Content-Length: 0" -i
HTTP/1.1 202 Accepted
Content-Type: application/json
Location:
https://myfunc.azurewebsites.net/runtime/webhooks/durabletask/instances/b79baf67f717453ca9e86c5da21e03ec

{"id": "b79baf67f717453ca9e86c5da21e03ec", ...}

> curl
https://myfunc.azurewebsites.net/runtime/webhooks/durabletask/instances/b79baf67f717453ca9e86c5da21e03ec -i
HTTP/1.1 202 Accepted
Content-Type: application/json
Location:
https://myfunc.azurewebsites.net/runtime/webhooks/durabletask/instances/b79baf67f717453ca9e86c5da21e03ec

{"runtimeStatus": "Running", "lastUpdatedTime": "2019-03-16T21:20:47Z", ...}

> curl
https://myfunc.azurewebsites.net/runtime/webhooks/durabletask/instances/b79baf67f717453ca9e86c5da21e03ec -i
HTTP/1.1 200 OK
Content-Length: 175
Content-Type: application/json

{"runtimeStatus": "Completed", "lastUpdatedTime": "2019-03-16T21:20:57Z", ...}

```

Because the Durable Functions runtime manages state for you, you don't need to implement your own status-tracking mechanism.

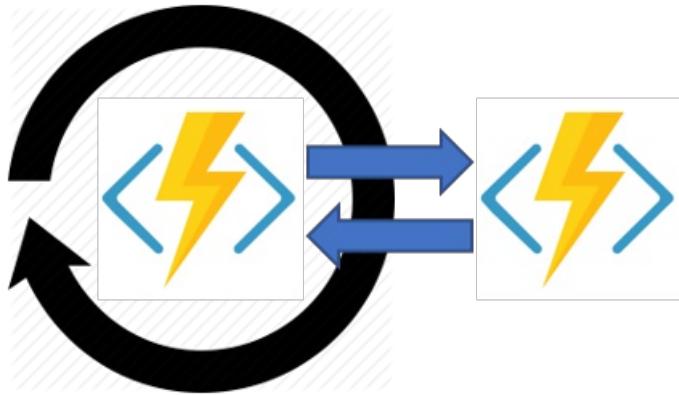
The Durable Functions extension exposes built-in HTTP APIs that manage long-running orchestrations. You can alternatively implement this pattern yourself by using your own function triggers (such as HTTP, a queue, or Azure Event Hubs) and the [orchestration client binding](#). For example, you might use a queue message to trigger termination. Or, you might use an HTTP trigger that's protected by an Azure Active Directory authentication policy instead of the built-in HTTP APIs that use a generated key for authentication.

For more information, see the [HTTP features](#) article, which explains how you can expose asynchronous, long-running processes over HTTP using the Durable Functions extension.

Pattern #4: Monitor

The monitor pattern refers to a flexible, recurring process in a workflow. An example is polling until specific conditions are met. You can use a regular [timer trigger](#) to address a basic scenario, such as a periodic cleanup job, but its interval is static and managing instance lifetimes becomes complex. You can use Durable Functions to create flexible recurrence intervals, manage task lifetimes, and create multiple monitor processes from a single orchestration.

An example of the monitor pattern is to reverse the earlier async HTTP API scenario. Instead of exposing an endpoint for an external client to monitor a long-running operation, the long-running monitor consumes an external endpoint, and then waits for a state change.



In a few lines of code, you can use Durable Functions to create multiple monitors that observe arbitrary endpoints. The monitors can end execution when a condition is met, or another function can use the durable orchestration client to terminate the monitors. You can change a monitor's `wait` interval based on a specific condition (for example, exponential backoff.)

The following code implements a basic monitor:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```
[FunctionName("MonitorJobStatus")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    int jobId = context.GetInput<int>();
    int pollingInterval = GetPollingInterval();
    DateTime expiryTime = GetExpiryTime();

    while (context.CurrentUtcDateTime < expiryTime)
    {
        var jobStatus = await context.CallActivityAsync<string>("GetJobStatus", jobId);
        if (jobStatus == "Completed")
        {
            // Perform an action when a condition is met.
            await context.CallActivityAsync("SendAlert", machineId);
            break;
        }

        // Orchestration sleeps until this time.
        var nextCheck = context.CurrentUtcDateTime.AddSeconds(pollingInterval);
        await context.CreateTimer(nextCheck, CancellationToken.None);
    }

    // Perform more work here, or let the orchestration end.
}
```

When a request is received, a new orchestration instance is created for that job ID. The instance polls a status until either a condition is met or until a timeout expires. A durable timer controls the polling interval. Then, more work can be performed, or the orchestration can end.

Pattern #5: Human interaction

Many automated processes involve some kind of human interaction. Involving humans in an automated process is tricky because people aren't as highly available and as responsive as cloud services. An automated process might allow for this interaction by using timeouts and compensation logic.

An approval process is an example of a business process that involves human interaction. Approval from a manager might be required for an expense report that exceeds a certain dollar amount. If the manager doesn't approve the expense report within 72 hours (maybe the manager went on vacation), an escalation process kicks in to get the approval from someone else (perhaps the manager's manager).



You can implement the pattern in this example by using an orchestrator function. The orchestrator uses a [durable timer](#) to request approval. The orchestrator escalates if timeout occurs. The orchestrator waits for an [external event](#), such as a notification that's generated by a human interaction.

These examples create an approval process to demonstrate the human interaction pattern:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```
[FunctionName("ApprovalWorkflow")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    await context.CallActivityAsync("RequestApproval", null);
    using (var timeoutCts = new CancellationTokenSource())
    {
        DateTime dueTime = context.CurrentUtcDateTime.AddHours(72);
        Task durableTimeout = context.CreateTimer(dueTime, timeoutCts.Token);

        Task<bool> approvalEvent = context.WaitForExternalEvent<bool>("ApprovalEvent");
        if (approvalEvent == await Task.WhenAny(approvalEvent, durableTimeout))
        {
            timeoutCts.Cancel();
            await context.CallActivityAsync("ProcessApproval", approvalEvent.Result);
        }
        else
        {
            await context.CallActivityAsync("Escalate", null);
        }
    }
}
```

To create the durable timer, call `context.CreateTimer`. The notification is received by `context.WaitForExternalEvent`. Then, `Task.WhenAny` is called to decide whether to escalate (timeout happens first) or process the approval (the approval is received before timeout).

NOTE

There is no charge for time spent waiting for external events when running in the Consumption plan.

An external client can deliver the event notification to a waiting orchestrator function by using the [built-in HTTP APIs](#):

```
curl -d "true"
http://localhost:7071/runtime/webhooks/durabletask/instances/{instanceId}/raiseEvent/ApprovalEvent -H
"Content-Type: application/json"
```

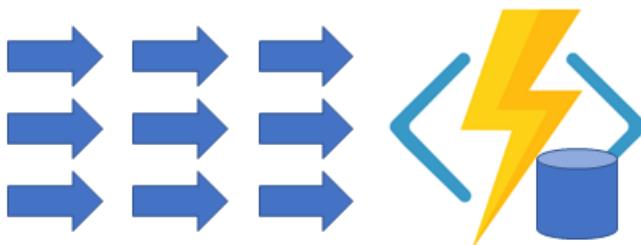
An event can also be raised using the durable orchestration client from another function in the same function app:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```
[FunctionName("RaiseEventToOrchestration")]
public static async Task Run(
    [HttpTrigger] string instanceId,
    [DurableClient] IDurableOrchestrationClient client)
{
    bool isApproved = true;
    await client.RaiseEventAsync(instanceId, "ApprovalEvent", isApproved);
}
```

Pattern #6: Aggregator (stateful entities)

The sixth pattern is about aggregating event data over a period of time into a single, addressable *entity*. In this pattern, the data being aggregated may come from multiple sources, may be delivered in batches, or may be scattered over long-periods of time. The aggregator might need to take action on event data as it arrives, and external clients may need to query the aggregated data.



The tricky thing about trying to implement this pattern with normal, stateless functions is that concurrency control becomes a huge challenge. Not only do you need to worry about multiple threads modifying the same data at the same time, you also need to worry about ensuring that the aggregator only runs on a single VM at a time.

You can use [Durable entities](#) to easily implement this pattern as a single function.

- [C#](#)

- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```
[FunctionName("Counter")]
public static void Counter([EntityTrigger] IDurableEntityContext ctx)
{
    int currentValue = ctx.GetState<int>();
    switch (ctx.OperationName.ToLowerInvariant())
    {
        case "add":
            int amount = ctx.GetInput<int>();
            ctx.SetState(currentValue + amount);
            break;
        case "reset":
            ctx.SetState(0);
            break;
        case "get":
            ctx.Return(currentValue);
            break;
    }
}
```

Durable entities can also be modeled as classes in .NET. This model can be useful if the list of operations is fixed and becomes large. The following example is an equivalent implementation of the `Counter` entity using .NET classes and methods.

```
public class Counter
{
    [JsonProperty("value")]
    public int CurrentValue { get; set; }

    public void Add(int amount) => this.CurrentValue += amount;

    public void Reset() => this.CurrentValue = 0;

    public int Get() => this.CurrentValue;

    [FunctionName(nameof(Counter))]
    public static Task Run([EntityTrigger] IDurableEntityContext ctx)
        => ctx.DispatchAsync<Counter>();
}
```

Clients can enqueue *operations* for (also known as "signaling") an entity function using the [entity client binding](#).

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```
[FunctionName("EventHubTriggerCSharp")]
public static async Task Run(
    [EventHubTrigger("device-sensor-events")] EventData eventData,
    [DurableClient] IDurableEntityClient entityClient)
{
    var metricType = (string)eventData.Properties["metric"];
    var delta = BitConverter.ToInt32(eventData.Body, eventData.Body.Offset);

    // The "Counter/{metricType}" entity is created on-demand.
    var entityId = new EntityId("Counter", metricType);
    await entityClient.SignalEntityAsync(entityId, "add", delta);
}
```

NOTE

Dynamically generated proxies are also available in .NET for signaling entities in a type-safe way. And in addition to signaling, clients can also query for the state of an entity function using [type-safe methods](#) on the orchestration client binding.

Entity functions are available in [Durable Functions 2.0](#) and above for C#, JavaScript, and Python.

The technology

Behind the scenes, the Durable Functions extension is built on top of the [Durable Task Framework](#), an open-source library on GitHub that's used to build workflows in code. Like Azure Functions is the serverless evolution of Azure WebJobs, Durable Functions is the serverless evolution of the Durable Task Framework. Microsoft and other organizations use the Durable Task Framework extensively to automate mission-critical processes. It's a natural fit for the serverless Azure Functions environment.

Code constraints

In order to provide reliable and long-running execution guarantees, orchestrator functions have a set of coding rules that must be followed. For more information, see the [Orchestrator function code constraints](#) article.

Billing

Durable Functions are billed the same as Azure Functions. For more information, see [Azure Functions pricing](#). When executing orchestrator functions in the Azure Functions [Consumption plan](#), there are some billing behaviors to be aware of. For more information on these behaviors, see the [Durable Functions billing](#) article.

Jump right in

You can get started with Durable Functions in under 10 minutes by completing one of these language-specific quickstart tutorials:

- [C# using Visual Studio 2019](#)
- [JavaScript using Visual Studio Code](#)
- [Python using Visual Studio Code](#)
- [PowerShell using Visual Studio Code](#)
- [Java using Maven](#)

In these quickstarts, you locally create and test a "hello world" durable function. You then publish the function code to Azure. The function you create orchestrates and chains together calls to other functions.

Publications

Durable Functions is developed in collaboration with Microsoft Research. As a result, the Durable Functions team actively produces research papers and artifacts; these include:

- [Durable Functions: Semantics for Stateful Serverless \(*OOPSLA'21*\)](#)
- [Serverless Workflows with Durable Functions and Netherite \(*pre-print*\)](#)

Learn more

The following video highlights the benefits of Durable Functions:

For a more in-depth discussion of Durable Functions and the underlying technology, see the following video (it's focused on .NET, but the concepts also apply to other supported languages):

Because Durable Functions is an advanced extension for [Azure Functions](#), it isn't appropriate for all applications.

For a comparison with other Azure orchestration technologies, see [Compare Azure Functions and Azure Logic Apps](#).

Next steps

[Durable Functions function types and features](#)

Choose the right integration and automation services in Azure

10/5/2022 • 6 minutes to read • [Edit Online](#)

This article compares the following Microsoft cloud services:

- [Microsoft Power Automate](#) (was Microsoft Flow)
- [Azure Logic Apps](#)
- [Azure Functions](#)
- [Azure App Service WebJobs](#)

All of these services can solve integration problems and automate business processes. They can all define input, actions, conditions, and output. You can run each of them on a schedule or trigger. Each service has unique advantages, and this article explains the differences.

If you're looking for a more general comparison between Azure Functions and other Azure compute options, see [Criteria for choosing an Azure compute service](#) and [Choosing an Azure compute option for microservices](#).

For a good summary and comparison of automation service options in Azure, see [Choose the Automation services in Azure](#).

Compare Microsoft Power Automate and Azure Logic Apps

Power Automate and Logic Apps are both *designer-first* integration services that can create workflows. Both services integrate with various SaaS and enterprise applications.

Power Automate is built on top of Logic Apps. They share the same workflow designer and the same [connectors](#).

Power Automate empowers any office worker to perform simple integrations (for example, an approval process on a SharePoint Document Library) without going through developers or IT. Logic Apps can also enable advanced integrations (for example, B2B processes) where enterprise-level Azure DevOps and security practices are required. It's typical for a business workflow to grow in complexity over time.

The following table helps you determine whether Power Automate or Logic Apps is best for a particular integration:

	POWER AUTOMATE	LOGIC APPS
Users	Office workers, business users, SharePoint administrators	Pro integrators and developers, IT pros
Scenarios	Self-service	Advanced integrations
Design tool	In-browser and mobile app, UI only	In-browser, Visual Studio Code , and Visual Studio with code view available
Application lifecycle management (ALM)	Design and test in non-production environments, promote to production when ready	Azure DevOps: source control, testing, support, automation, and manageability in Azure Resource Manager

	POWER AUTOMATE	LOGIC APPS
Admin experience	Manage Power Automate environments and data loss prevention (DLP) policies, track licensing: Admin center	Manage resource groups, connections, access management, and logging: Azure portal
Security	Microsoft 365 security audit logs, DLP, encryption at rest for sensitive data	Security assurance of Azure: Azure security , Microsoft Defender for Cloud , audit logs

Compare Azure Functions and Azure Logic Apps

Functions and Logic Apps are Azure services that enable serverless workloads. Azure Functions is a serverless compute service, whereas Azure Logic Apps is a serverless workflow integration platform. Both can create complex *orchestrations*. An orchestration is a collection of functions, or *actions* in Azure Logic Apps, that you can run to complete a complex task. For example, to process a batch of orders, you might execute many instances of a function in parallel, wait for all instances to finish, and then execute a function that computes a result on the aggregate.

For Azure Functions, you develop orchestrations by writing code and using the [Durable Functions extension](#). For Azure Logic Apps, you create orchestrations by using a GUI or editing configuration files.

You can mix and match services when you build an orchestration, such as calling functions from logic app workflows and calling logic app workflows from functions. Choose how to build each orchestration based on the services' capabilities or your personal preference. The following table lists some key differences between these services:

	DURABLE FUNCTIONS	LOGIC APPS
Development	Code-first (imperative)	Designer-first (declarative)
Connectivity	About a dozen built-in binding types , write code for custom bindings	Large collection of connectors , Enterprise Integration Pack for B2B scenarios , build custom connectors
Actions	Each activity is an Azure function; write code for activity functions	Large collection of ready-made actions
Monitoring	Azure Application Insights	Azure portal , Azure Monitor logs , Microsoft Defender for Cloud
Management	REST API , Visual Studio	Azure portal , REST API , PowerShell , Visual Studio
Execution context	Can run locally or in the cloud	Runs in Azure, locally, or on premises. For more information, see What is Azure Logic Apps .

Compare Functions and WebJobs

Like Azure Functions, Azure App Service WebJobs with the WebJobs SDK is a *code-first* integration service that is designed for developers. Both are built on [Azure App Service](#) and support features such as [source control integration](#), [authentication](#), and [monitoring with Application Insights integration](#).

WebJobs and the WebJobs SDK

You can use the *WebJobs* feature of App Service to run a script or code in the context of an App Service web app. The *WebJobs SDK* is a framework designed for WebJobs that simplifies the code you write to respond to events in Azure services. For example, you might respond to the creation of an image blob in Azure Storage by creating a thumbnail image. The WebJobs SDK runs as a .NET console application, which you can deploy to a WebJob.

WebJobs and the WebJobs SDK work best together, but you can use WebJobs without the WebJobs SDK and vice versa. A WebJob can run any program or script that runs in the App Service sandbox. A WebJobs SDK console application can run anywhere console applications run, such as on-premises servers.

Comparison table

Azure Functions is built on the WebJobs SDK, so it shares many of the same event triggers and connections to other Azure services. Here are some factors to consider when you're choosing between Azure Functions and WebJobs with the WebJobs SDK:

	FUNCTIONS	WEBJOBS WITH WEBJOBS SDK
Serverless app model with automatic scaling	✓	
Develop and test in browser	✓	
Pay-per-use pricing	✓	
Integration with Logic Apps	✓	
Trigger events	Timer Azure Storage queues and blobs Azure Service Bus queues and topics Azure Cosmos DB Azure Event Hubs HTTP/WebHook (GitHub, Slack) Azure Event Grid	Timer Azure Storage queues and blobs Azure Service Bus queues and topics Azure Cosmos DB Azure Event Hubs File system
Supported languages	C# F# JavaScript Java Python PowerShell	C# ¹
Package managers	NPM and NuGet	NuGet ²

¹ WebJobs (without the WebJobs SDK) supports languages such as C#, Java, JavaScript, Bash, .cmd, .bat, PowerShell, PHP, TypeScript, Python, and more. A WebJob can run any program or script that can run in the App Service sandbox.

² WebJobs (without the WebJobs SDK) supports NPM and NuGet.

Summary

Azure Functions offers more developer productivity than Azure App Service WebJobs does. It also offers more options for programming languages, development environments, Azure service integration, and pricing. For most scenarios, it's the best choice.

Here are two scenarios for which WebJobs may be the best choice:

- You need more control over the code that listens for events, the `JobHost` object. Functions offers a limited number of ways to customize `JobHost` behavior in the `host.json` file. Sometimes you need to do things that can't be specified by a string in a JSON file. For example, only the WebJobs SDK lets you configure a custom retry policy for Azure Storage.
- You have an App Service app for which you want to run code snippets, and you want to manage them together in the same Azure DevOps environment.

For other scenarios where you want to run code snippets for integrating Azure or third-party services, choose Azure Functions over WebJobs with the WebJobs SDK.

Power Automate, Logic Apps, Functions, and WebJobs together

You don't have to choose just one of these services. They integrate with each other as well as with external services.

A Power Automate flow can call an Azure Logic Apps workflow. An Azure Logic Apps workflow can call a function in Azure Functions, and vice versa. For example, see [Create a function that integrates with Azure Logic Apps](#).

Between Power Automate, Logic Apps, and Functions, the integration experience between these services continues to improve over time. You can build a component in one service and use that component in the other services.

You can get more information on integration services by using the following links:

- [Leveraging Azure Functions & Azure App Service for integration scenarios by Christopher Anderson](#)
- [Integrations Made Simple by Charles Lamanna](#)
- [Logic Apps Live webcast](#)
- [Power Automate frequently asked questions](#)

Next steps

Get started by creating your first flow, logic app workflow, or function app. Select any of the following links:

- [Get started with Power Automate](#)
- [Create your first logic app workflow](#)
- [Create your first Azure function](#)

Durable Functions versions overview

10/5/2022 • 3 minutes to read • [Edit Online](#)

Durable Functions is an extension of [Azure Functions](#) and [Azure WebJobs](#) that lets you write stateful functions in a serverless environment. The extension manages state, checkpoints, and restarts for you. If you aren't already familiar with Durable Functions, see the [overview documentation](#).

New features in 2.x

This section describes the features of Durable Functions that are added in version 2.x.

Durable entities

In Durable Functions 2.x, we introduced a new [entity functions](#) concept.

Entity functions define operations for reading and updating small pieces of state, known as *durable entities*. Like orchestrator functions, entity functions are functions with a special trigger type, *entity trigger*. Unlike orchestrator functions, entity functions don't have any specific code constraints. Entity functions also manage state explicitly rather than implicitly representing state via control flow.

To learn more, see the [durable entities](#) article.

Durable HTTP

In Durable Functions 2.x, we introduced a new [Durable HTTP](#) feature that allows you to:

- Call HTTP APIs directly from orchestration functions (with some documented limitations).
- Implement automatic client-side HTTP 202 status polling.
- Built-in support for [Azure Managed Identities](#).

To learn more, see the [HTTP features](#) article.

Migrate from 1.x to 2.x

This section describes how to migrate your existing version 1.x Durable Functions to version 2.x to take advantage of the new features.

Upgrade the extension

Install the latest 2.x version of the Durable Functions bindings extension in your project.

JavaScript, Python, and PowerShell

Durable Functions 2.x is available starting in version 2.x of the [Azure Functions extension bundle](#).

Python support in Durable Functions requires Durable Functions 2.x or greater.

To update the extension bundle version in your project, open host.json and update the `extensionBundle` section to use version 3.x (`[3.*, 4.0.0)`).

```
{  
    "version": "2.0",  
    "extensionBundle": {  
        "id": "Microsoft.Azure.Functions.ExtensionBundle",  
        "version": "[3.*, 4.0.0)"  
    }  
}
```

NOTE

If Visual Studio Code is not displaying the correct templates after you change the extension bundle version, reload the window by running the *Developer: Reload Window* command (Ctrl+R on Windows and Linux, Command+R on macOS).

Java (preview)

Durable Functions 2.x is available starting in version 4.x of the [Azure Functions extension bundle](#). You must use the Azure Functions 3.0 runtime or greater to execute Java functions.

To update the extension bundle version in your project, open host.json and update the `extensionBundle` section to use version 4.x (`[4.* , 5.0.0)`). Because Java support is currently in preview, you must also use the `Microsoft.Azure.Functions.ExtensionBundle.Preview` bundle, which is different from product-ready bundles.

```
{  
    "version": "2.0",  
    "extensionBundle": {  
        "id": "Microsoft.Azure.Functions.ExtensionBundle.Preview",  
        "version": "[4.* , 5.0.0)"  
    }  
}
```

.NET

Update your .NET project to use the latest version of the [Durable Functions bindings extension](#).

See [Register Azure Functions binding extensions](#) for more information.

Update your code

Durable Functions 2.x introduces several breaking changes. Durable Functions 1.x applications aren't compatible with Durable Functions 2.x without code changes. This section lists some of the changes you must make when upgrading your version 1.x functions to 2.x.

Host.json schema

Durable Functions 2.x uses a new host.json schema. The main changes from 1.x include:

- `"storageProvider"` (and the `"azureStorage"` subsection) for storage-specific configuration.
- `"tracing"` for tracing and logging configuration.
- `"notifications"` (and the `"eventGrid"` subsection) for Event Grid notification configuration.

See the [Durable Functions host.json reference documentation](#) for details.

Default task hub name changes

In version 1.x, if a task hub name wasn't specified in host.json, it was defaulted to "DurableFunctionsHub". In version 2.x, the default task hub name is now derived from the name of the function app. Because of this, if you haven't specified a task hub name when upgrading to 2.x, your code will be operating with new task hub, and all in-flight orchestrations will no longer have an application processing them. To work around this, you can either explicitly set your task hub name to the v1.x default of "DurableFunctionsHub", or you can follow our [zero-downtime deployment guidance](#) for details on how to handle breaking changes for in-flight orchestrations.

Public interface changes (.NET only)

In version 1.x, the various *context* objects supported by Durable Functions have abstract base classes intended for use in unit testing. As part of Durable Functions 2.x, these abstract base classes are replaced with interfaces.

The following table represents the main changes:

1.X	2.X
DurableOrchestrationClientBase	IDurableOrchestrationClient or IDurableClient
DurableOrchestrationContext or DurableOrchestrationContextBase	IDurableOrchestrationContext
DurableActivityContext or DurableActivityContextBase	IDurableActivityContext
OrchestrationClientAttribute	DurableClientAttribute

In the case where an abstract base class contained virtual methods, these virtual methods have been replaced by extension methods defined in `DurableContextExtensions`.

function.json changes (JavaScript and C# Script)

In Durable Functions 1.x, the orchestration client binding uses a `type` of `orchestrationClient`. Version 2.x uses `durableClient` instead.

Raise event changes

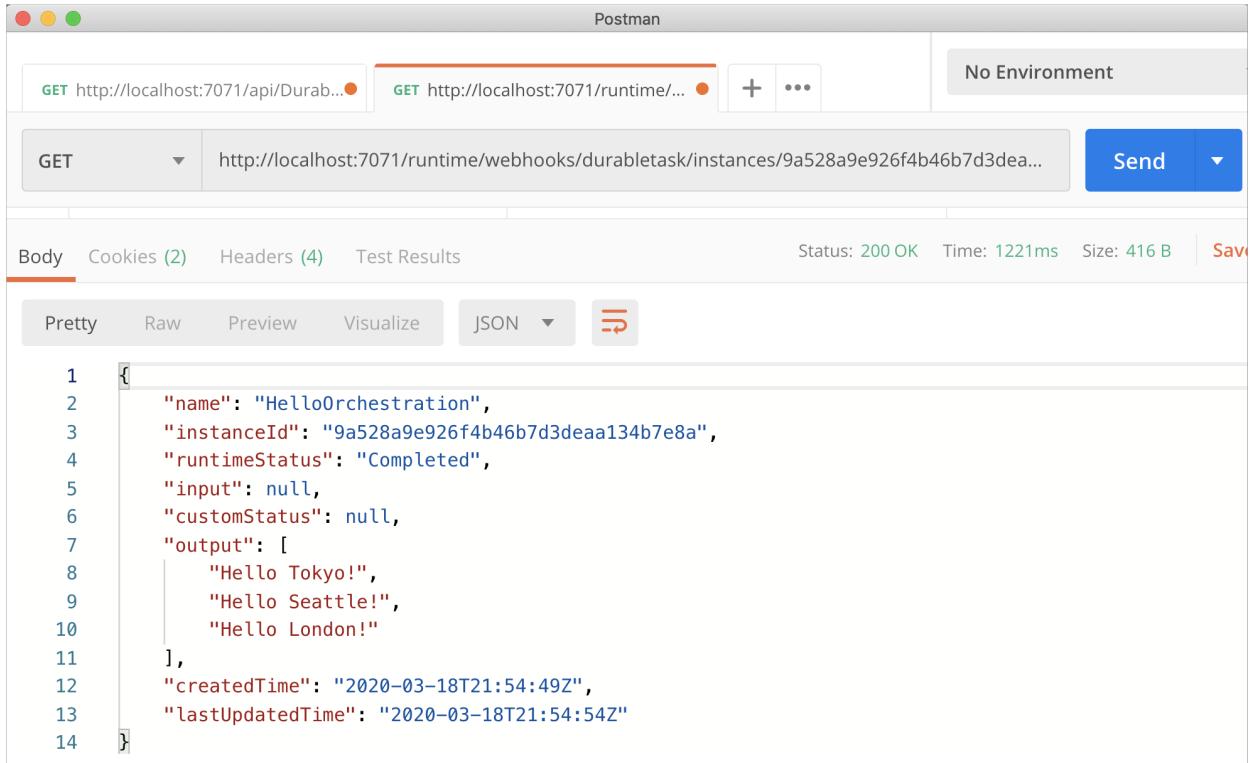
In Durable Functions 1.x, calling the [raise event](#) API and specifying an instance that didn't exist resulted in a silent failure. Starting in 2.x, raising an event to a non-existent orchestration results in an exception.

Create your first durable function in C#

10/5/2022 • 14 minutes to read • [Edit Online](#)

Durable Functions is an extension of [Azure Functions](#) that lets you write stateful functions in a serverless environment. The extension manages state, checkpoints, and restarts for you.

In this article, you learn how to use Visual Studio Code to locally create and test a "hello world" durable function. This function orchestrates and chains together calls to other functions. You can then publish the function code to Azure. These tools are available as part of the Visual Studio Code [Azure Functions extension](#).



Prerequisites

To complete this tutorial:

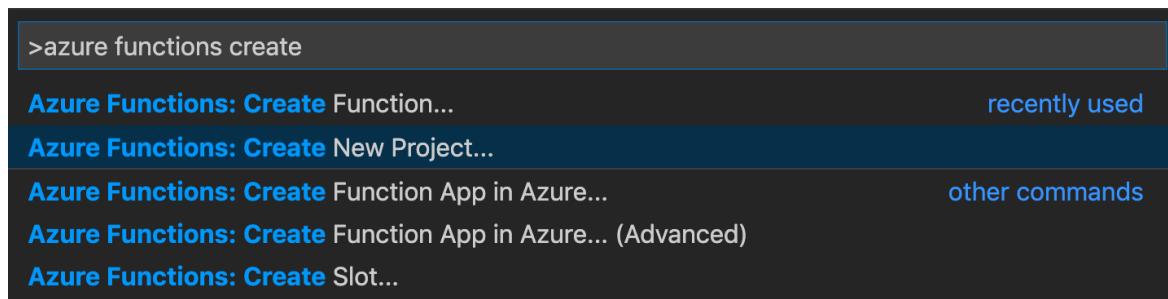
- Install [Visual Studio Code](#).
- Install the following Visual Studio Code extensions:
 - [Azure Functions](#)
 - [C#](#)
- Make sure that you have the latest version of the [Azure Functions Core Tools](#).
- Durable Functions require an Azure storage account. You need an Azure subscription.
- Make sure that you have version 3.1 or a later version of the [.NET Core SDK](#) installed.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Create your local project

In this section, you use Visual Studio Code to create a local Azure Functions project.

1. In Visual Studio Code, press F1 (or Ctrl/Cmd+Shift+P) to open the command palette. In the command palette, search for and select `Azure Functions: Create New Project...`.



2. Choose an empty folder location for your project and choose **Select**.

3. Follow the prompts and provide the following information:

PROMPT	VALUE	DESCRIPTION
Select a language for your function app project	C#	Create a local C# Functions project.
Select a version	Azure Functions v4	You only see this option when the Core Tools aren't already installed. In this case, Core Tools are installed the first time you run the app.
Select a template for your project's first function	Skip for now	
Select how you would like to open your project	Open in current window	Reopens Visual Studio Code in the folder you selected.

Visual Studio Code installs the Azure Functions Core Tools if needed. It also creates a function app project in a folder. This project contains the `host.json` and `local.settings.json` configuration files.

Add functions to the app

The following steps use a template to create the durable function code in your project.

1. In the command palette, search for and select `Azure Functions: Create Function...`.

2. Follow the prompts and provide the following information:

PROMPT	VALUE	DESCRIPTION
Select a template for your function	DurableFunctionsOrchestration	Create a Durable Functions orchestration
Provide a function name	HelloOrchestration	Name of the class in which functions are created
Provide a namespace	Company.Function	Namespace for the generated class

3. When Visual Studio Code prompts you to select a storage account, choose **Select storage account**.

Follow the prompts and provide the following information to create a new storage account in Azure:

PROMPT	VALUE	DESCRIPTION
Select subscription	<i>name of your subscription</i>	Select your Azure subscription
Select a storage account	Create a new storage account	
Enter the name of the new storage account	<i>unique name</i>	Name of the storage account to create
Select a resource group	<i>unique name</i>	Name of the resource group to create
Select a location	<i>region</i>	Select a region close to you

A class containing the new functions is added to the project. Visual Studio Code also adds the storage account connection string to `local.settings.json` and a reference to the [Microsoft.Azure.WebJobs.Extensions.DurableTask](#) NuGet package to the `.csproj` project file.

Open the new `HelloOrchestration.cs` file to view the contents. This durable function is a simple function chaining example with the following methods:

METHOD	FUNCTIONNAME	DESCRIPTION
<code>RunOrchestrator</code>	<code>HelloOrchestration</code>	Manages the durable orchestration. In this case, the orchestration starts, creates a list, and adds the result of three functions calls to the list. When the three function calls are complete, it returns the list.
<code>SayHello</code>	<code>HelloOrchestration_Hello</code>	The function returns a hello. It's the function that contains the business logic that is being orchestrated.
<code>HttpStart</code>	<code>HelloOrchestration_HttpStart</code>	An HTTP-triggered function that starts an instance of the orchestration and returns a check status response.

Now that you've created your function project and a durable function, you can test it on your local computer.

Test the function locally

Azure Functions Core Tools lets you run an Azure Functions project on your local development computer. You're prompted to install these tools the first time you start a function from Visual Studio Code.

- To test your function, set a breakpoint in the `SayHello` activity function code and press F5 to start the function app project. Output from Core Tools is displayed in the **Terminal** panel.

NOTE

For more information on debugging, see [Durable Functions Diagnostics](#).

- In the **Terminal** panel, copy the URL endpoint of your HTTP-triggered function.

The screenshot shows the Visual Studio Code interface with the Terminal tab selected. The output window displays log messages from the host initialization process. One message highlights the URL endpoint for the 'HelloOrchestration_HttpStart' function: `http://localhost:7071/api/HelloOrchestration_HttpStart`.

3. Use a tool like [Postman](#) or [cURL](#), and then send an HTTP POST request to the URL endpoint.

The response is the HTTP function's initial result, letting us know that the durable orchestration has started successfully. It isn't yet the end result of the orchestration. The response includes a few useful URLs. For now, let's query the status of the orchestration.

4. Copy the URL value for `statusQueryGetUri`, paste it into the browser's address bar, and execute the request. Alternatively, you can also continue to use Postman to issue the GET request.

The request will query the orchestration instance for the status. You must get an eventual response, which shows us that the instance has completed and includes the outputs or results of the durable function. It looks like:

```
{  
  "name": "HelloOrchestration",  
  "instanceId": "9a528a9e926f4b46b7d3deaa134b7e8a",  
  "runtimeStatus": "Completed",  
  "input": null,  
  "customStatus": null,  
  "output": [  
    "Hello Tokyo!",  
    "Hello Seattle!",  
    "Hello London!"  
  ],  
  "createdTime": "2020-03-18T21:54:49Z",  
  "lastUpdatedTime": "2020-03-18T21:54:54Z"  
}
```

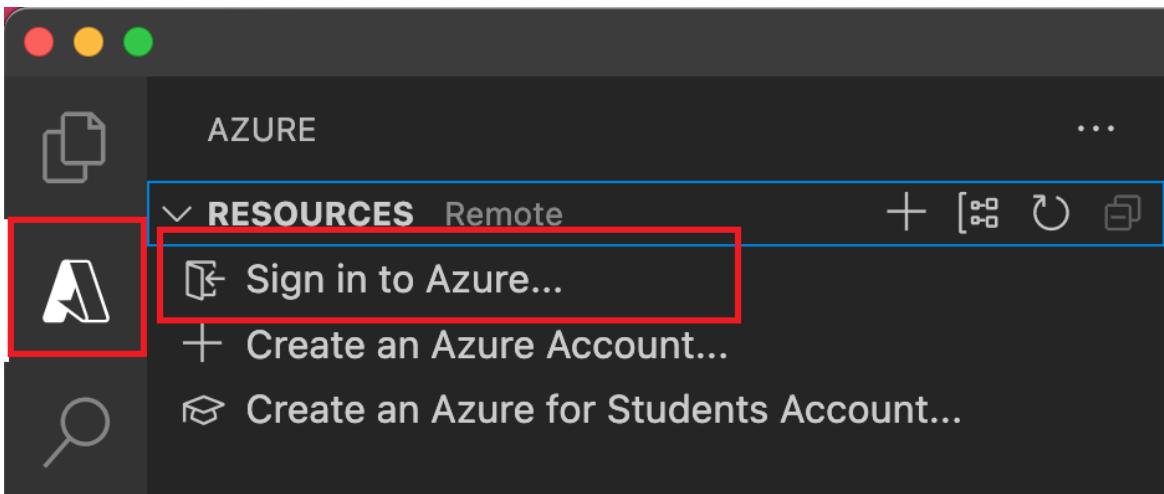
5. To stop debugging, press Shift + F5 in Visual Studio Code.

After you've verified that the function runs correctly on your local computer, it's time to publish the project to Azure.

Sign in to Azure

Before you can publish your app, you must sign in to Azure.

1. If you aren't already signed in, choose the Azure icon in the Activity bar. Then in the Resources area, choose **Sign in to Azure....**



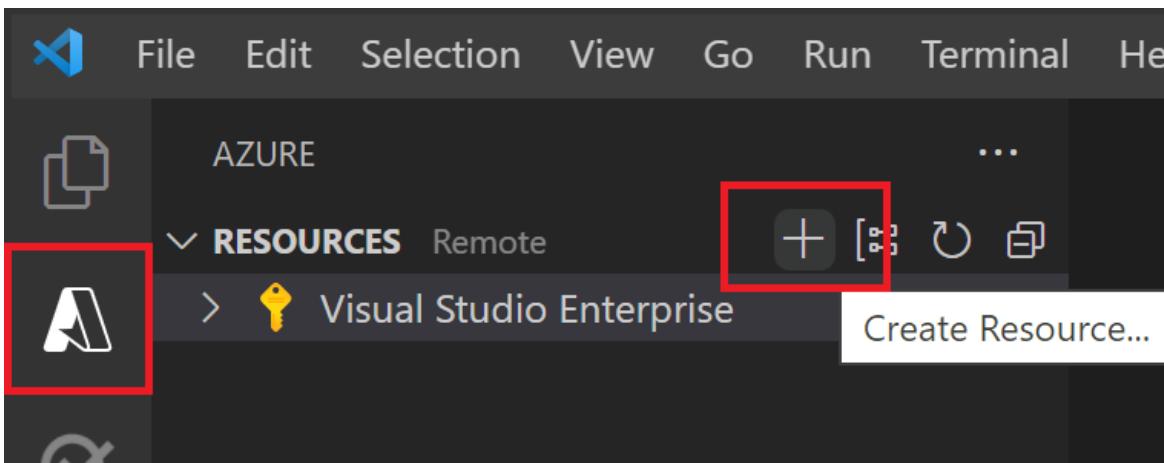
If you're already signed in and can see your existing subscriptions, go to the next section. If you don't yet have an Azure account, choose **Create an Azure Account....** Students can choose **Create an Azure for Students Account....**

2. When prompted in the browser, choose your Azure account and sign in using your Azure account credentials. If you create a new account, you can sign in after your account is created.
3. After you've successfully signed in, you can close the new browser window. The subscriptions that belong to your Azure account are displayed in the sidebar.

Create the function app in Azure

In this section, you create a function app and related resources in your Azure subscription.

1. Choose the Azure icon in the Activity bar. Then in the Resources area, select the + icon and choose the **Create Function App in Azure** option.

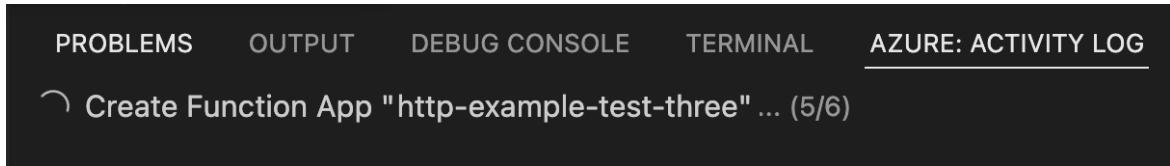


2. Provide the following information at the prompts:

PROMPT	SELECTION
Select subscription	Choose the subscription to use. You won't see this prompt when you have only one subscription visible under Resources .
Enter a globally unique name for the function app	Type a name that is valid in a URL path. The name you type is validated to make sure that it's unique in Azure Functions.

PROMPT	SELECTION
Select a runtime stack	Choose the language version on which you've been running locally.
Select a location for new resources	For better performance, choose a region near you.

The extension shows the status of individual resources as they're being created in Azure in the [Azure: Activity Log](#) panel.



- When the creation is complete, the following Azure resources are created in your subscription. The resources are named based on your function app name:
 - A [resource group](#), which is a logical container for related resources.
 - A standard [Azure Storage account](#), which maintains state and other information about your projects.
 - A function app, which provides the environment for executing your function code. A function app lets you group functions as a logical unit for easier management, deployment, and sharing of resources within the same hosting plan.
 - An App Service plan, which defines the underlying host for your function app.
 - An Application Insights instance connected to the function app, which tracks usage of your functions in the app.

A notification is displayed after your function app is created and the deployment package is applied.

TIP

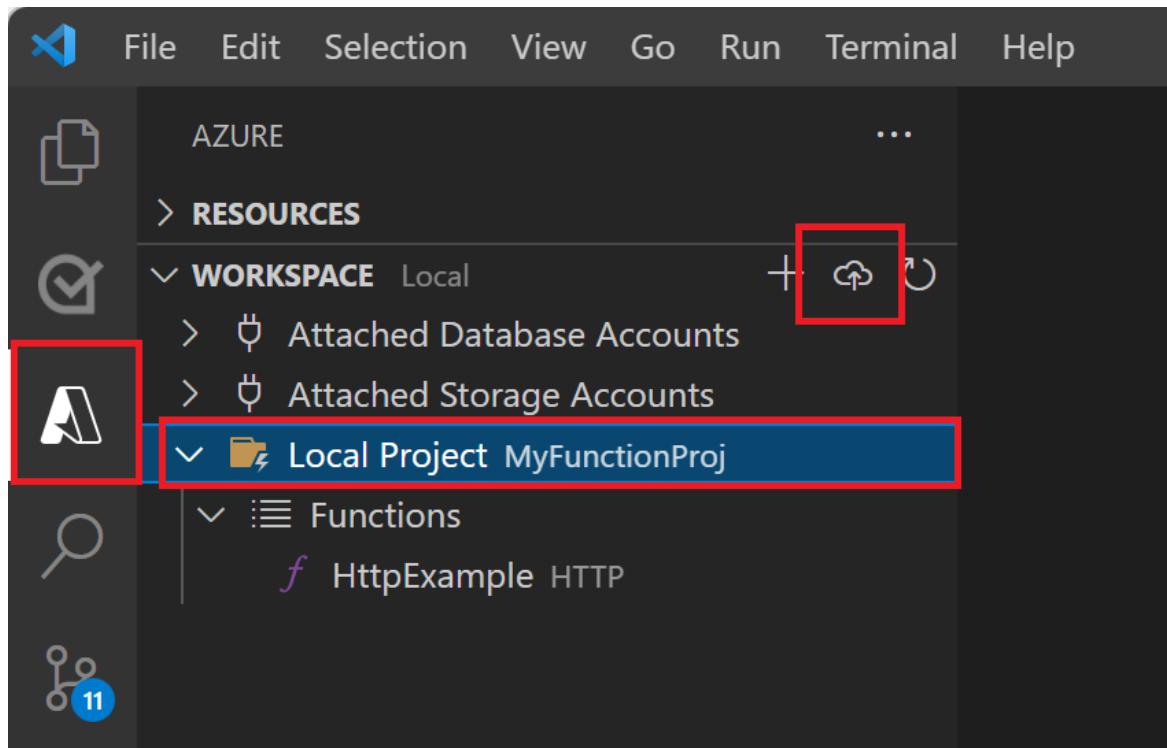
By default, the Azure resources required by your function app are created based on the function app name you provide. By default, they're also created in the same new resource group with the function app. If you want to either customize the names of these resources or reuse existing resources, you need to [publish the project with advanced create options](#) instead.

Deploy the project to Azure

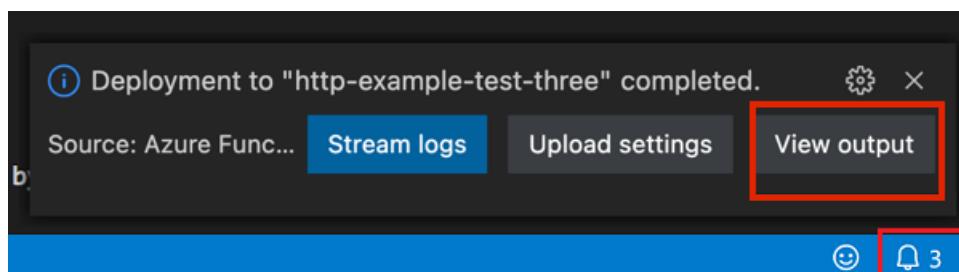
IMPORTANT

Deploying to an existing function app always overwrites the contents of that app in Azure.

- Choose the Azure icon in the Activity bar, then in the **Workspace** area, select your project folder and select the **Deploy...** button.



2. Select **Deploy to Function App...**, choose the function app you just created, and select **Deploy**.
3. After deployment completes, select **View Output** to view the creation and deployment results, including the Azure resources that you created. If you miss the notification, select the bell icon in the lower right corner to see it again.



Test your function in Azure

1. Copy the URL of the HTTP trigger from the **Output** panel. The URL that calls your HTTP-triggered function must be in the following format:

```
https://<functionappname>.azurewebsites.net/api/HelloOrchestration_HttpStart
```

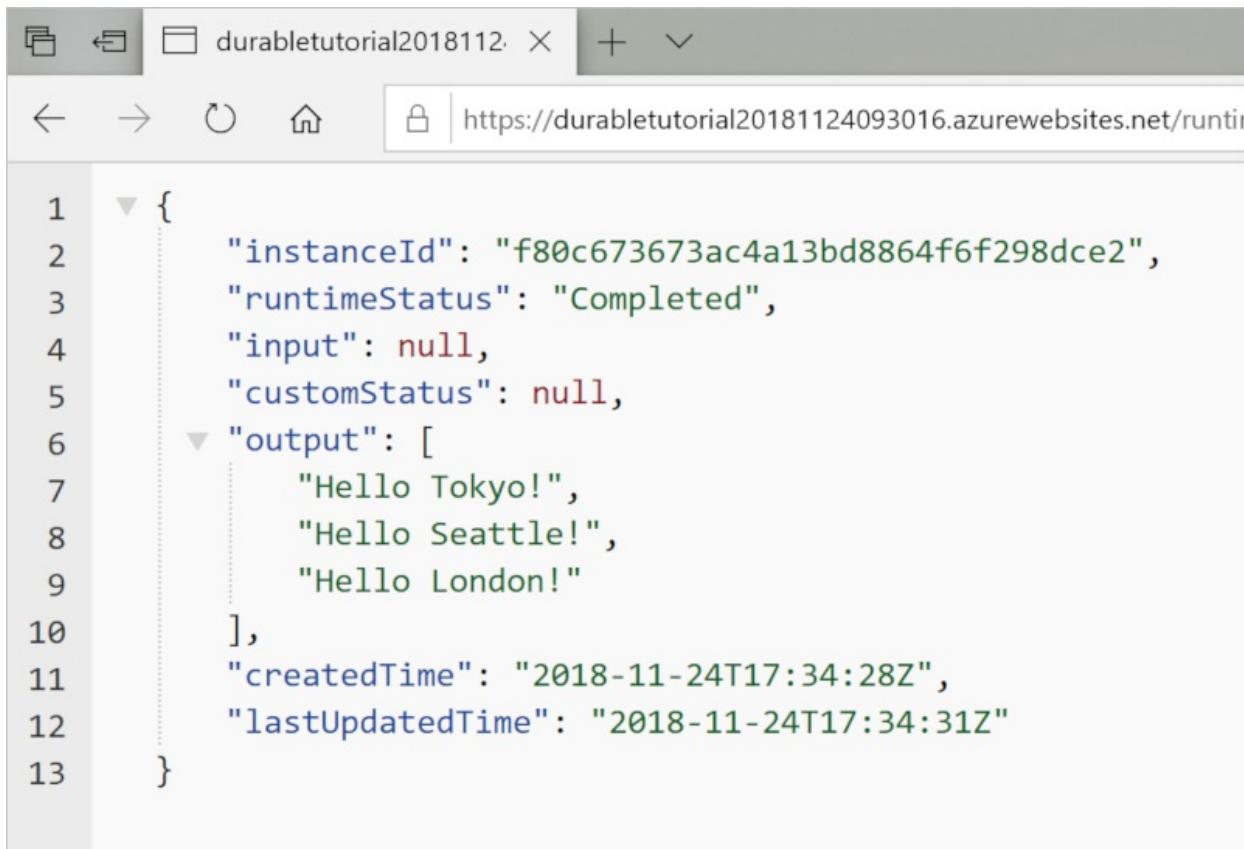
2. Paste this new URL for the HTTP request into your browser's address bar. You must get the same status response as before when using the published app.

Next steps

You have used Visual Studio Code to create and publish a C# durable function app.

[Learn about common durable function patterns](#)

In this article, you learn how to use Visual Studio 2022 to locally create and test a "hello world" durable function. This function orchestrates and chains-together calls to other functions. You then publish the function code to Azure. These tools are available as part of the Azure development workload in Visual Studio 2022.



The screenshot shows a browser window with the title bar "durabletutorial2018112... X". The address bar contains the URL "https://durabletutorial20181124093016.azurewebsites.net/runtir". The main content area displays a JSON object with the following structure:

```
1  {
2      "instanceId": "f80c673673ac4a13bd8864f6f298dce2",
3      "runtimeStatus": "Completed",
4      "input": null,
5      "customStatus": null,
6      "output": [
7          "Hello Tokyo!",
8          "Hello Seattle!",
9          "Hello London!"
10     ],
11     "createdTime": "2018-11-24T17:34:28Z",
12     "lastUpdatedTime": "2018-11-24T17:34:31Z"
13 }
```

Prerequisites

To complete this tutorial:

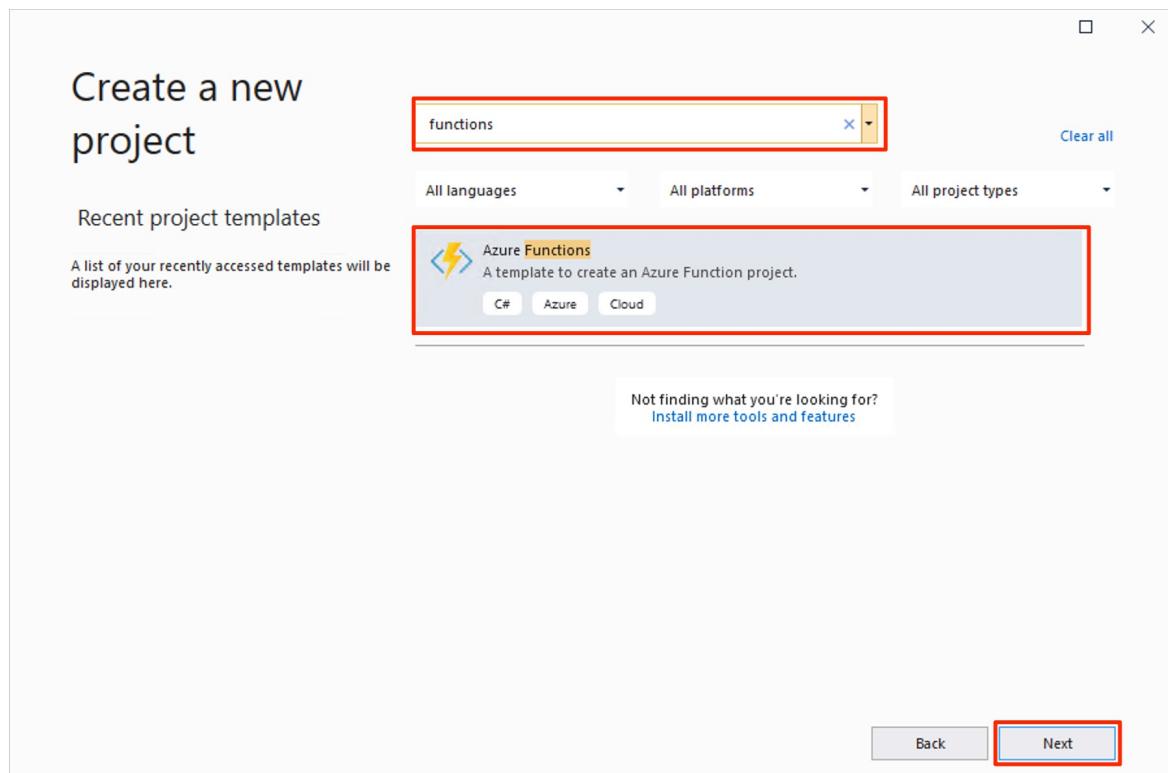
- Install [Visual Studio 2022](#). Make sure that the **Azure development** workload is also installed. Visual Studio 2019 also supports Durable Functions development, but the UI and steps differ.
- Verify that you have the [Azurite Emulator](#) installed and running.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

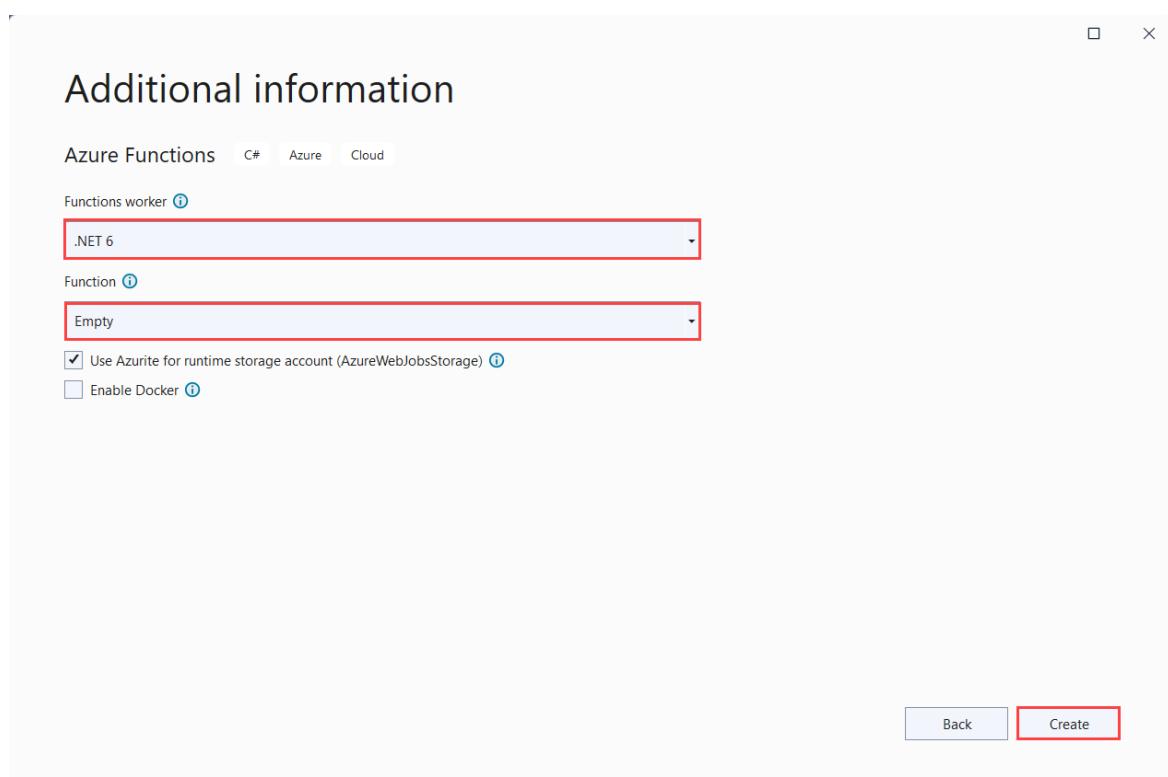
Create a function app project

The Azure Functions template creates a project that can be published to a function app in Azure. A function app lets you group functions as a logical unit for easier management, deployment, scaling, and sharing of resources.

1. In Visual Studio, select **New > Project** from the **File** menu.
2. In the **Create a new project** dialog, search for `functions`, choose the **Azure Functions** template, and then select **Next**.



3. Enter a **Project name** for your project, and select **OK**. The project name must be valid as a C# namespace, so don't use underscores, hyphens, or nonalphanumeric characters.
4. Under **Additional information**, use the settings specified in the table that follows the image.



SETTING	SUGGESTED VALUE	DESCRIPTION
Functions worker	.NET 6	Creates a function project that supports .NET 6 and the Azure Functions Runtime 4.0. For more information, see How to target Azure Functions runtime version .

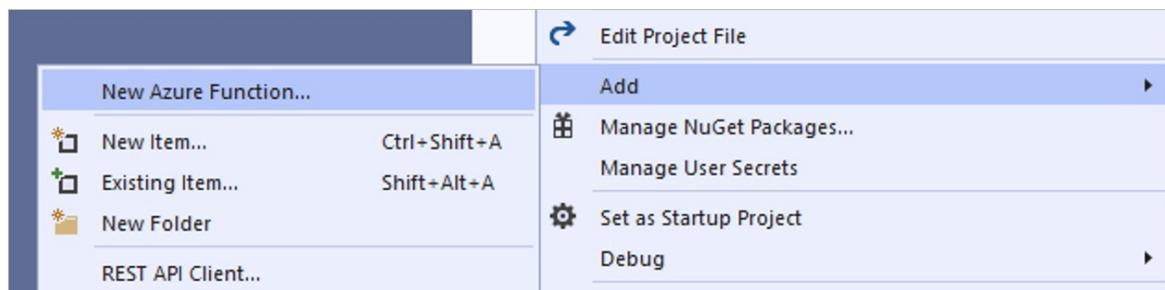
SETTING	SUGGESTED VALUE	DESCRIPTION
Function	Empty	Creates an empty function app.
Storage account	Storage Emulator	A storage account is required for durable function state management.

5. Select **Create** to create an empty function project. This project has the basic configuration files needed to run your functions.

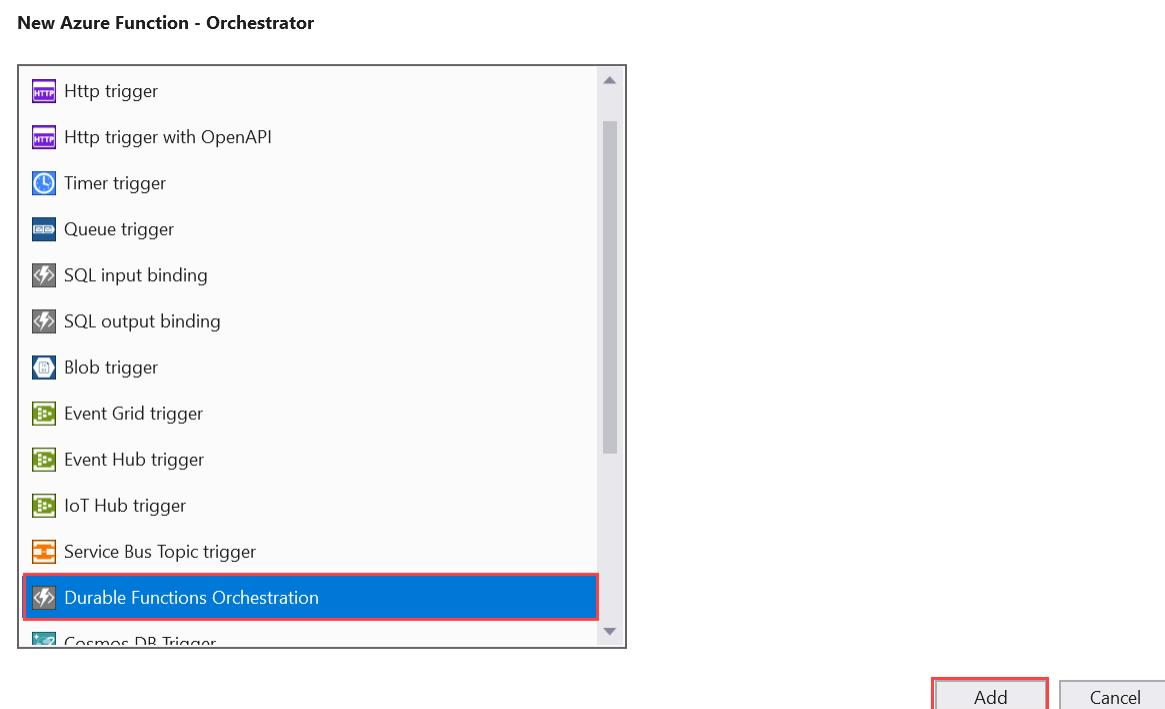
Add functions to the app

The following steps use a template to create the durable function code in your project.

1. Right-click the project in Visual Studio and select **Add > New Azure Function**.



2. Verify **Azure Function** is selected from the add menu, enter a name for your C# file, and then select **Add**.
3. Select the **Durable Functions Orchestration** template and then select **Add**.



A new durable function is added to the app. Open the new .cs file to view the contents. This durable function is a simple function chaining example with the following methods:

METHOD	FUNCTIONNAME	DESCRIPTION

METHOD	FUNCTIONNAME	DESCRIPTION
RunOrchestrator	<file-name>	Manages the durable orchestration. In this case, the orchestration starts, creates a list, and adds the result of three functions calls to the list. When the three function calls are complete, it returns the list.
SayHello	<file-name>_Hello	The function returns a hello. It's the function that contains the business logic that is being orchestrated.
HttpStart	<file-name>_HttpStart	An HTTP-triggered function that starts an instance of the orchestration and returns a check status response.

You can test it on your local computer now that you've created your function project and a durable function.

Test the function locally

Azure Functions Core Tools lets you run an Azure Functions project on your local development computer. You're prompted to install these tools the first time you start a function from Visual Studio.

1. To test your function, press F5. If prompted, accept the request from Visual Studio to download and install Azure Functions Core (CLI) tools. You may also need to enable a firewall exception so that the tools can handle HTTP requests.
2. Copy the URL of your function from the Azure Functions runtime output.

```
Name: . ExtensionVersion: 1.6.2. SequenceNumber: 1.
[11/8/2018 7:05:31 AM] Host started (2709ms)
[11/8/2018 7:05:31 AM] Job host started
Hosting environment: Production
Content root path: C:\Users\jeffhollan\source\repos\DurableTutorial\DurableTutorial\bin\Debug\netcoreapp2.1
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.
Listening on http://0.0.0.0:7071/
Hit CTRL-C to exit...

Http Functions:

Durable_HttpStart: [GET,POST] http://localhost:7071/api/Durable_HttpStart
[11/8/2018 7:05:38 AM] Host lock lease acquired by instance ID '0000000000000000000000000000000075A5073E'.
```

3. Paste the URL for the HTTP request into your browser's address bar and execute the request. The following shows the response in the browser to the local GET request returned by the function:



```
{"id": "d495cb0ac10d4e13b22729c37e335190", "statusQueryGetUri": "http://localhost:7071/runtime/webhooks/durabletask/instances/d495cb0ac10d4e13b22729c37e335190?"}
```

The response is the HTTP function's initial result, letting us know that the durable orchestration has started successfully. It isn't yet the end result of the orchestration. The response includes a few useful URLs. For now, let's query the status of the orchestration.

4. Copy the URL value for `statusQueryGetUri`, paste it into the browser's address bar, and execute the request.

The request will query the orchestration instance for the status. You must get an eventual response that looks like the following. This output shows us the instance has completed and includes the outputs or results of the durable function.

```
{  
    "name": "Durable",  
    "instanceId": "d495cb0ac10d4e13b22729c37e335190",  
    "runtimeStatus": "Completed",  
    "input": null,  
    "customStatus": null,  
    "output": [  
        "Hello Tokyo!",  
        "Hello Seattle!",  
        "Hello London!"  
    ],  
    "createdTime": "2019-11-02T07:07:40Z",  
    "lastUpdatedTime": "2019-11-02T07:07:52Z"  
}
```

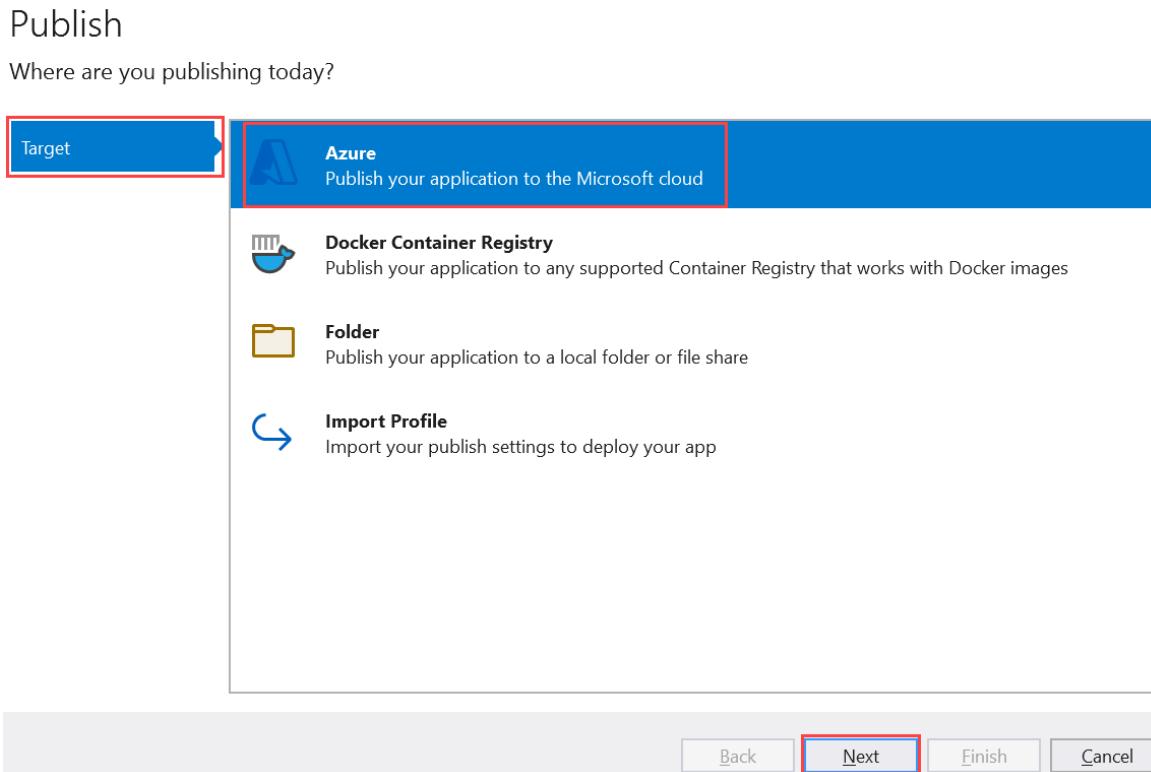
5. To stop debugging, press Shift + F5.

After you've verified that the function runs correctly on your local computer, it's time to publish the project to Azure.

Publish the project to Azure

You must have a function app in your Azure subscription before publishing your project. You can create a function app right from Visual Studio.

1. In **Solution Explorer**, right-click the project and select **Publish**. In **Target**, select **Azure** then **Next**.



2. Select **Azure Function App (Windows)** for the **Specific target**, which creates a function app that runs on Windows, and then select **Next**.

Publish

Which Azure service would you like to use to host your application?

The screenshot shows the 'Publish' dialog with the 'Target' section expanded. A red box highlights the 'Specific target' button. Below it, four options are listed:

- Azure Function App (Windows)**: Publish your application code to a serverless compute that scales dynamically and runs code on-demand.
- Azure Function App (Linux)**: Publish your application code to a serverless compute that scales dynamically and runs code on-demand.
- Azure Function App Container**: Publish your application as a Docker image to Azure Container Registry and run it on Azure Function App.
- Azure Container Registry**: Publish your application as a Docker image to Azure Container Registry.

At the bottom are 'Back', 'Next' (highlighted with a red box), 'Finish', and 'Cancel' buttons.

3. In the **Function Instance**, choose **Create a new Azure Function...**

The screenshot shows the 'Publish' dialog with the 'Target' section expanded. A red box highlights the 'Functions instance' button. On the right, a sidebar shows 'Function Apps' with three items: 'functiondemodeploy', 'LogicAppResourceGroup', and 'NewDocs'. A red box highlights the '+ Create a new Azure Function...' button. At the bottom are 'Back', 'Next' (highlighted with a red box), 'Finish', and 'Cancel' buttons.

4. Create a new instance using the values specified in the following table:

SETTING	VALUE	DESCRIPTION
Name	Globally unique name	Name that uniquely identifies your new function app. Accept this name or enter a new name. Valid characters are: a-z, 0-9, and -.

SETTING	VALUE	DESCRIPTION
Subscription	Your subscription	The Azure subscription to use. Accept this subscription or select a new one from the drop-down list.
Resource group	Name of your resource group	The resource group in which you want to create your function app. Select an existing resource group from the drop-down list or select New to create a new resource group.
Plan Type	Consumption	When you publish your project to a function app that runs in a Consumption plan , you pay only for executions of your functions app. Other hosting plans incur higher costs.
Location	Location of the app service	Choose a Location in a region near you or other services your functions access.
Azure Storage	General-purpose storage account	An Azure storage account is required by the Functions runtime. Select New to configure a general-purpose storage account. You can also choose an existing account that meets the storage account requirements .

 Function App (Windows)
Create new

Name
FunctionApp220200930124040

Subscription
Visual Studio Enterprise

Resource group
azure-docs-first-function* [New...](#)

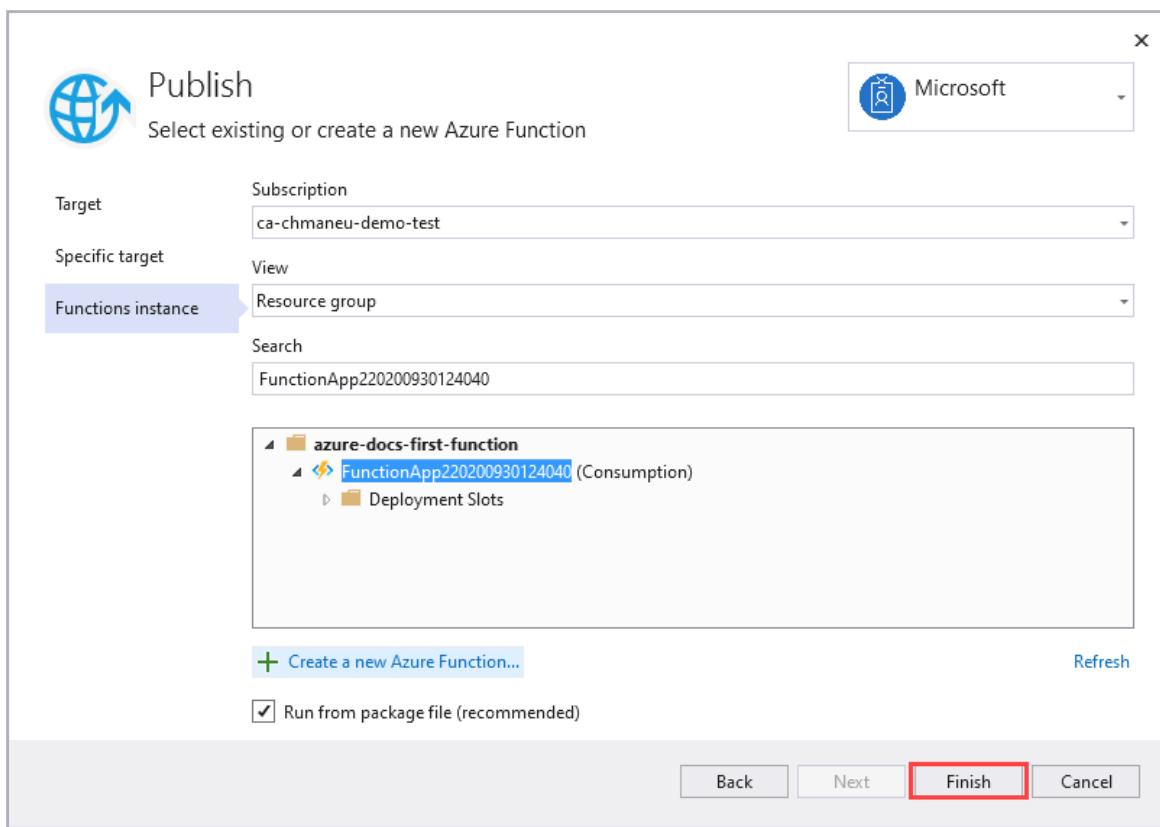
Plan Type
Consumption

Location
France Central

Azure Storage
functiondemochris* (France Central) [New...](#)

Create Create **Cancel**

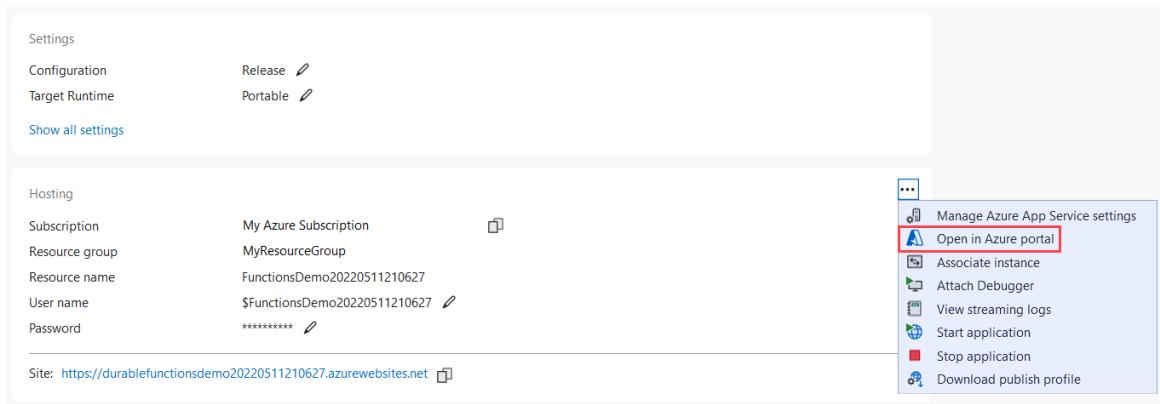
- Select **Create** to create a function app and its related resources in Azure. The status of resource creation is shown in the lower-left of the window.
- In the **Functions instance**, make sure that the **Run from package file** is checked. Your function app is deployed using **Zip Deploy** with **Run-From-Package** mode enabled. Zip Deploy is the recommended deployment method for your functions project resulting in better performance.



- Select **Finish**, and on the Publish page, select **Publish** to deploy the package containing your project files to your new function app in Azure.

After the deployment completes, the root URL of the function app in Azure is shown in the **Publish** tab.

- In the Publish tab, in the Hosting section, choose **Open in Azure portal**. This opens the new function app Azure resource in the Azure portal.



Test your function in Azure

- Copy the base URL of the function app from the Publish profile page. Replace the `localhost:port` portion of the URL you used when testing the function locally with the new base URL.

The URL that calls your durable function HTTP trigger must be in the following format:

`https://<APP_NAME>.azurewebsites.net/api/<FUNCTION_NAME>_HttpStart`

2. Paste this new URL for the HTTP request into your browser's address bar. You must get the same status response as before when using the published app.

Next steps

You have used Visual Studio to create and publish a C# durable function app.

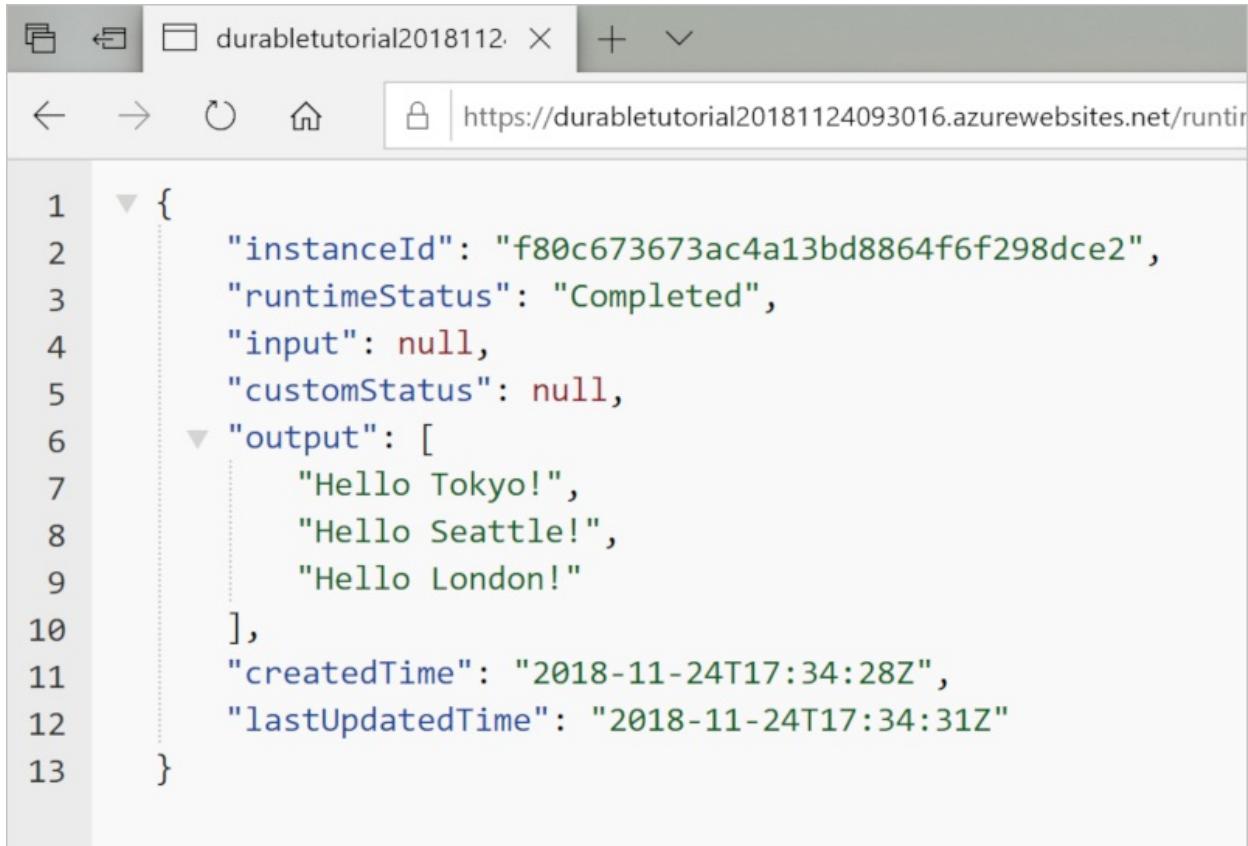
[Learn about common durable function patterns](#)

Create your first durable function in JavaScript

10/5/2022 • 9 minutes to read • [Edit Online](#)

Durable Functions is an extension of [Azure Functions](#) that lets you write stateful functions in a serverless environment. The extension manages state, checkpoints, and restarts for you.

In this article, you learn how to use the Visual Studio Code Azure Functions extension to locally create and test a "hello world" durable function. This function will orchestrate and chain together calls to other functions. You then publish the function code to Azure.



The screenshot shows a browser window with the URL <https://durabletutorial20181124093016.azurewebsites.net/runtime>. The page displays a JSON object representing the state of a durable function instance. The JSON is as follows:

```
1  {
2      "instanceId": "f80c673673ac4a13bd8864f6f298dce2",
3      "runtimeStatus": "Completed",
4      "input": null,
5      "customStatus": null,
6      "output": [
7          "Hello Tokyo!",
8          "Hello Seattle!",
9          "Hello London!"
10     ],
11     "createdTime": "2018-11-24T17:34:28Z",
12     "lastUpdatedTime": "2018-11-24T17:34:31Z"
13 }
```

Prerequisites

To complete this tutorial:

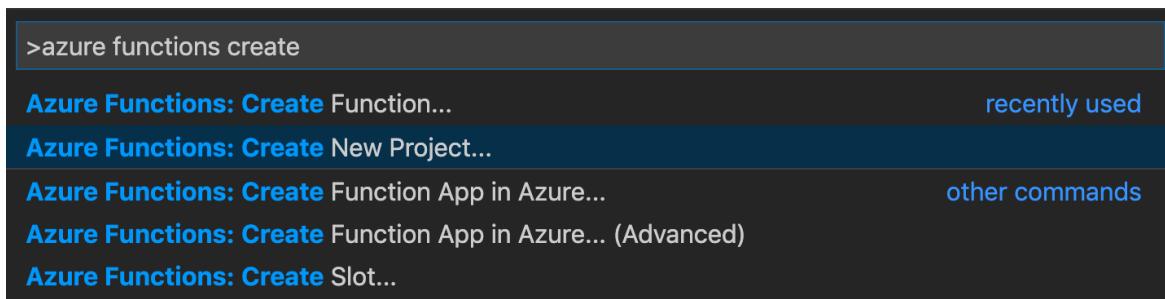
- Install [Visual Studio Code](#).
- Install the [Azure Functions](#) VS Code extension
- Make sure you have the latest version of the [Azure Functions Core Tools](#).
- Durable Functions require an Azure storage account. You need an Azure subscription.
- Make sure that you have version 10.x or 12.x of [Node.js](#) installed.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Create your local project

In this section, you use Visual Studio Code to create a local Azure Functions project.

1. In Visual Studio Code, press F1 (or Ctrl/Cmd+Shift+P) to open the command palette. In the command palette, search for and select `Azure Functions: Create New Project...`.



2. Choose an empty folder location for your project and choose **Select**.

3. Following the prompts, provide the following information:

PROMPT	VALUE	DESCRIPTION
Select a language for your function app project	JavaScript	Create a local Node.js Functions project.
Select a version	Azure Functions v4	You only see this option when the Core Tools aren't already installed. In this case, Core Tools are installed the first time you run the app.
Select a template for your project's first function	Skip for now	
Select how you would like to open your project	Open in current window	Reopens VS Code in the folder you selected.

Visual Studio Code installs the Azure Functions Core Tools, if needed. It also creates a function app project in a folder. This project contains the `host.json` and `local.settings.json` configuration files.

A `package.json` file is also created in the root folder.

Install the Durable Functions npm package

To work with Durable Functions in a Node.js function app, you use a library called `durable-functions`.

1. Use the *View* menu or **Ctrl+Shift+`** to open a new terminal in VS Code.
2. Install the `durable-functions` npm package by running `npm install durable-functions` in the root directory of the function app.

Creating your functions

The most basic Durable Functions app contains three functions:

- *Orchestrator function* - describes a workflow that orchestrates other functions.
- *Activity function* - called by the orchestrator function, performs work, and optionally returns a value.
- *Client function* - a regular Azure Function that starts an orchestrator function. This example uses an HTTP triggered function.

Orchestrator function

You use a template to create the durable function code in your project.

1. In the command palette, search for and select `Azure Functions: Create Function...`.

2. Following the prompts, provide the following information:

PROMPT	VALUE	DESCRIPTION
Select a template for your function	Durable Functions orchestrator	Create a Durable Functions orchestration
Provide a function name	HelloOrchestrator	Name of your durable function

You've added an orchestrator to coordinate activity functions. Open `HelloOrchestrator/index.js` to see the orchestrator function. Each call to `context.df.callActivity` invokes an activity function named `Hello`.

Next, you'll add the referenced `Hello` activity function.

Activity function

1. In the command palette, search for and select `Azure Functions: Create Function...`.

2. Following the prompts, provide the following information:

PROMPT	VALUE	DESCRIPTION
Select a template for your function	Durable Functions activity	Create an activity function
Provide a function name	Hello	Name of your activity function

You've added the `Hello` activity function that is invoked by the orchestrator. Open `Hello/index.js` to see that it's taking a name as input and returning a greeting. An activity function is where you'll perform actions such as making a database call or performing a computation.

Finally, you'll add an HTTP triggered function that starts the orchestration.

Client function (HTTP starter)

1. In the command palette, search for and select `Azure Functions: Create Function...`.

2. Following the prompts, provide the following information:

PROMPT	VALUE	DESCRIPTION
Select a template for your function	Durable Functions HTTP starter	Create an HTTP starter function
Provide a function name	DurableFunctionsHttpStart	Name of your activity function
Authorization level	Anonymous	For demo purposes, allow the function to be called without authentication

You've added an HTTP triggered function that starts an orchestration. Open `DurableFunctionsHttpStart/index.js` to see that it uses `client.startNew` to start a new orchestration. Then it uses `client.createCheckStatusResponse` to return an HTTP response containing URLs that can be used to monitor and manage the new orchestration.

You now have a Durable Functions app that can be run locally and deployed to Azure.

Test the function locally

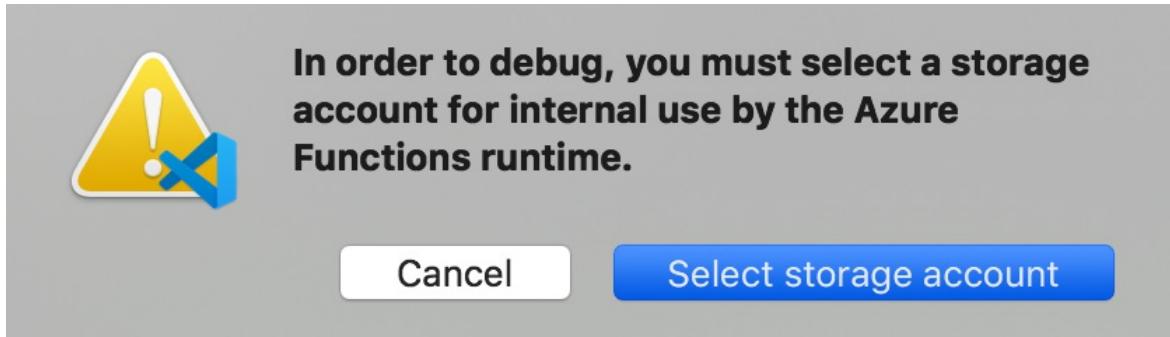
Azure Functions Core Tools lets you run an Azure Functions project on your local development computer. You're prompted to install these tools the first time you start a function from Visual Studio Code.

1. To test your function, set a breakpoint in the `Hello` activity function code (`Hello/index.js`). Press F5 or select `Debug: Start Debugging` from the command palette to start the function app project. Output from Core Tools is displayed in the `Terminal` panel.

NOTE

Refer to the [Durable Functions Diagnostics](#) for more information on debugging.

2. Durable Functions requires an Azure Storage account to run. When VS Code prompts you to select a storage account, choose **Select storage account**.



3. Following the prompts, provide the following information to create a new storage account in Azure.

PROMPT	VALUE	DESCRIPTION
Select subscription	<i>name of your subscription</i>	Select your Azure subscription
Select a storage account	Create a new storage account	
Enter the name of the new storage account	<i>unique name</i>	Name of the storage account to create
Select a resource group	<i>unique name</i>	Name of the resource group to create
Select a location	<i>region</i>	Select a region close to you

4. In the **Terminal** panel, copy the URL endpoint of your HTTP-triggered function.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 2: Task - runFunctionsH ▾ + ⚡ 🗑 ⏷ ⏸ ×  
Now listening on: http://0.0.0.0:7071  
Application started. Press Ctrl+C to shut down.  
  
Http Functions:  
  
HttpStart: [POST] http://localhost:7071/api/orchestrators/{functionName}
```

5. Using your browser, or a tool like [Postman](#) or [cURL](#), send an HTTP POST request to the URL endpoint.

Replace the last segment with the name of the orchestrator function (`HelloOrchestrator`). The URL should be similar to HelloOrchestrator.

The response is the initial result from the HTTP function letting you know the durable orchestration has started successfully. It is not yet the end result of the orchestration. The response includes a few useful

URLs. For now, let's query the status of the orchestration.

6. Copy the URL value for `statusQueryGetUri` and paste it in the browser's address bar and execute the request. Alternatively you can also continue to use Postman to issue the GET request.

The request will query the orchestration instance for the status. You should get an eventual response, which shows us the instance has completed, and includes the outputs or results of the durable function. It looks like:

```
{  
  "name": "HelloOrchestrator",  
  "instanceId": "9a528a9e926f4b46b7d3deaa134b7e8a",  
  "runtimeStatus": "Completed",  
  "input": null,  
  "customStatus": null,  
  "output": [  
    "Hello Tokyo!",  
    "Hello Seattle!",  
    "Hello London!"  
  ],  
  "createdTime": "2020-03-18T21:54:49Z",  
  "lastUpdatedTime": "2020-03-18T21:54:54Z"  
}
```

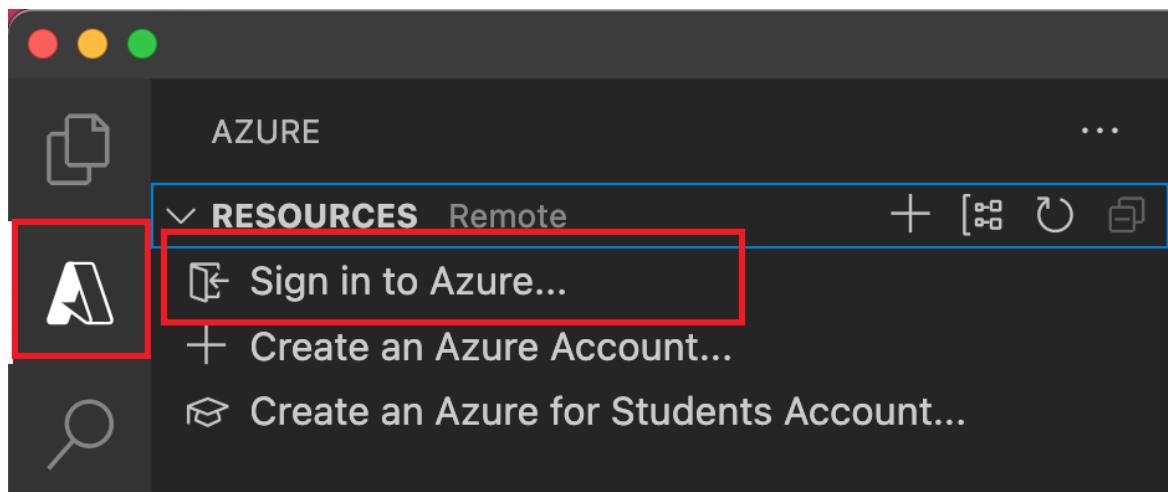
7. To stop debugging, press **Shift + F5** in VS Code.

After you've verified that the function runs correctly on your local computer, it's time to publish the project to Azure.

Sign in to Azure

Before you can publish your app, you must sign in to Azure.

1. If you aren't already signed in, choose the Azure icon in the Activity bar. Then in the **Resources** area, choose **Sign in to Azure....**



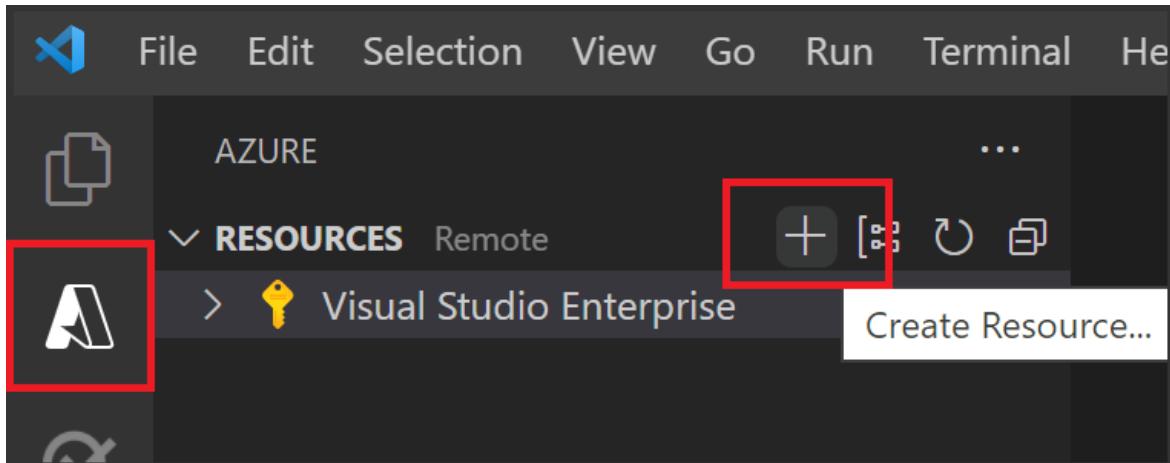
If you're already signed in and can see your existing subscriptions, go to the next section. If you don't yet have an Azure account, choose **Create an Azure Account....** Students can choose **Create an Azure for Students Account....**

2. When prompted in the browser, choose your Azure account and sign in using your Azure account credentials. If you create a new account, you can sign in after your account is created.
3. After you've successfully signed in, you can close the new browser window. The subscriptions that belong to your Azure account are displayed in the sidebar.

Create the function app in Azure

In this section, you create a function app and related resources in your Azure subscription.

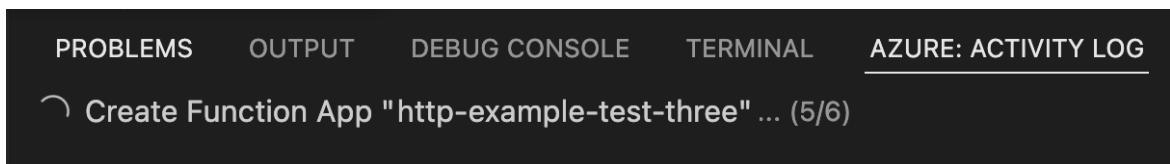
1. Choose the Azure icon in the Activity bar. Then in the Resources area, select the + icon and choose the **Create Function App in Azure** option.



2. Provide the following information at the prompts:

PROMPT	SELECTION
Select subscription	Choose the subscription to use. You won't see this prompt when you have only one subscription visible under Resources .
Enter a globally unique name for the function app	Type a name that is valid in a URL path. The name you type is validated to make sure that it's unique in Azure Functions.
Select a runtime stack	Choose the language version on which you've been running locally.
Select a location for new resources	For better performance, choose a region near you.

The extension shows the status of individual resources as they're being created in Azure in the **Azure: Activity Log** panel.



3. When the creation is complete, the following Azure resources are created in your subscription. The resources are named based on your function app name:

- A [resource group](#), which is a logical container for related resources.
- A standard [Azure Storage account](#), which maintains state and other information about your projects.
- A function app, which provides the environment for executing your function code. A function app lets you group functions as a logical unit for easier management, deployment, and sharing of resources within the same hosting plan.
- An App Service plan, which defines the underlying host for your function app.
- An Application Insights instance connected to the function app, which tracks usage of your functions in the app.

A notification is displayed after your function app is created and the deployment package is applied.

TIP

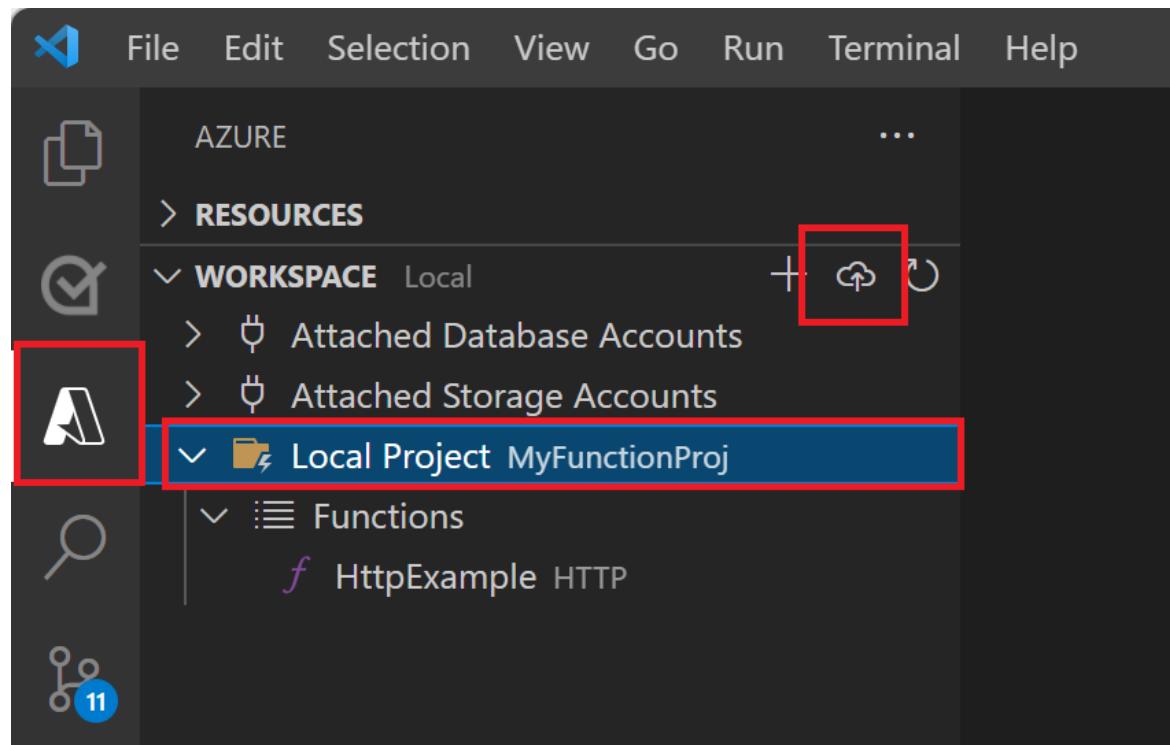
By default, the Azure resources required by your function app are created based on the function app name you provide. By default, they're also created in the same new resource group with the function app. If you want to either customize the names of these resources or reuse existing resources, you need to [publish the project with advanced create options](#) instead.

Deploy the project to Azure

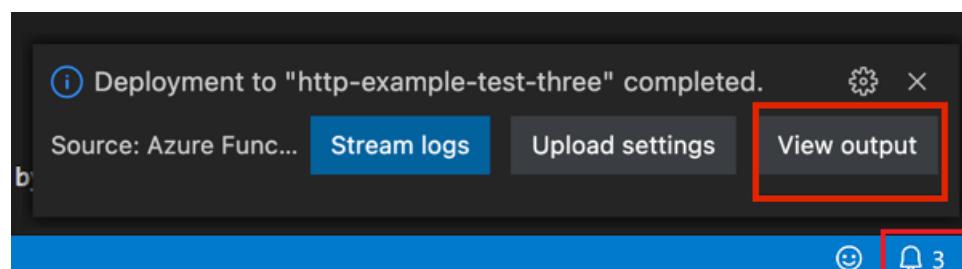
IMPORTANT

Deploying to an existing function app always overwrites the contents of that app in Azure.

1. Choose the Azure icon in the Activity bar, then in the **Workspace** area, select your project folder and select the **Deploy...** button.



2. Select **Deploy to Function App...**, choose the function app you just created, and select **Deploy**.
3. After deployment completes, select **View Output** to view the creation and deployment results, including the Azure resources that you created. If you miss the notification, select the bell icon in the lower right corner to see it again.



Test your function in Azure

1. Copy the URL of the HTTP trigger from the **Output** panel. The URL that calls your HTTP-triggered function should be in this format:
`http://<functionappname>.azurewebsites.net/api/orchestrators/HelloOrchestrator`
2. Paste this new URL for the HTTP request into your browser's address bar. You should get the same status response as before when using the published app.

Next steps

You have used Visual Studio Code to create and publish a JavaScript durable function app.

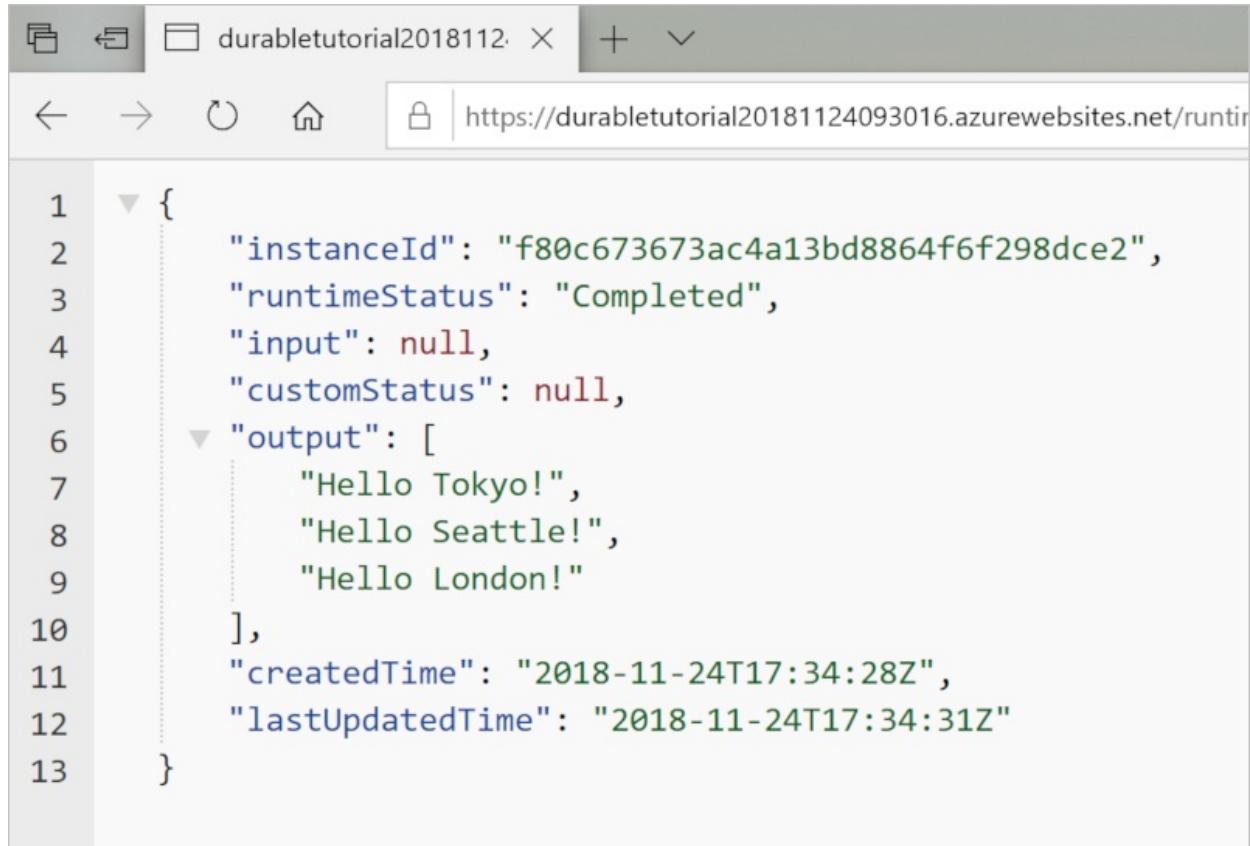
[Learn about common durable function patterns](#)

Create your first durable function in Python

10/5/2022 • 9 minutes to read • [Edit Online](#)

Durable Functions is an extension of [Azure Functions](#) that lets you write stateful functions in a serverless environment. The extension manages state, checkpoints, and restarts for you.

In this article, you learn how to use the Visual Studio Code Azure Functions extension to locally create and test a "hello world" durable function. This function will orchestrate and chains together calls to other functions. You can then publish the function code to Azure.



The screenshot shows a browser window with the URL <https://durabletutorial20181124093016.azurewebsites.net/runtime>. The page displays a JSON object representing the state of a durable function instance. The JSON is as follows:

```
1  {
2      "instanceId": "f80c673673ac4a13bd8864f6f298dce2",
3      "runtimeStatus": "Completed",
4      "input": null,
5      "customStatus": null,
6      "output": [
7          "Hello Tokyo!",
8          "Hello Seattle!",
9          "Hello London!"
10     ],
11     "createdTime": "2018-11-24T17:34:28Z",
12     "lastUpdatedTime": "2018-11-24T17:34:31Z"
13 }
```

Prerequisites

To complete this tutorial:

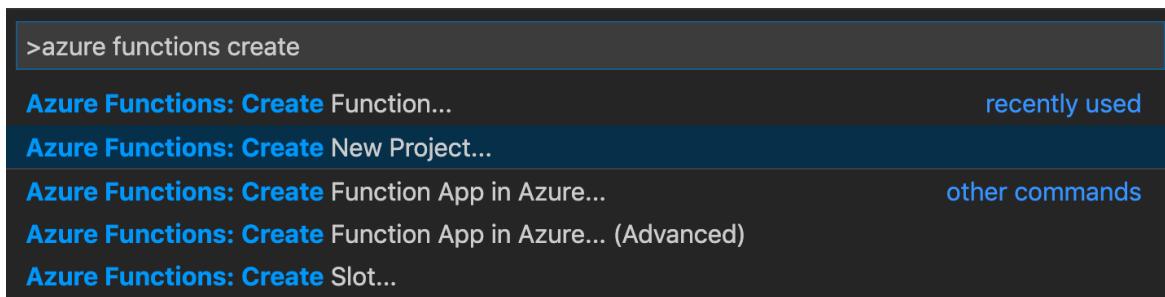
- Install [Visual Studio Code](#).
- Install the [Azure Functions](#) Visual Studio Code extension.
- Make sure that you have the latest version of the [Azure Functions Core Tools](#).
- Durable Functions require an Azure storage account. You need an Azure subscription.
- Make sure that you have version 3.7, 3.8, or 3.9 of [Python](#) installed.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Create your local project

In this section, you use Visual Studio Code to create a local Azure Functions project.

1. In Visual Studio Code, press F1 (or Ctrl/Cmd+Shift+P) to open the command palette. In the command palette, search for and select `Azure Functions: Create New Project...`.



2. Choose an empty folder location for your project and choose **Select**.

3. Follow the prompts and provide the following information:

PROMPT	VALUE	DESCRIPTION
Select a language for your function app project	Python	Create a local Python Functions project.
Select a version	Azure Functions v4	You only see this option when the Core Tools aren't already installed. In this case, Core Tools are installed the first time you run the app.
Python version	Python 3.6, 3.7, or 3.8	Visual Studio Code will create a virtual environment with the version you select.
Select a template for your project's first function	Skip for now	
Select how you would like to open your project	Open in current window	Reopens Visual Studio Code in the folder you selected.

Visual Studio Code installs the Azure Functions Core Tools if needed. It also creates a function app project in a folder. This project contains the `host.json` and `local.settings.json` configuration files.

A `requirements.txt` file is also created in the root folder. It specifies the Python packages required to run your function app.

Install azure-functions-durable from PyPI

When you've created the project, the Azure Functions Visual Studio Code extension automatically creates a virtual environment with your selected Python version. You then need to activate the virtual environment in a terminal and install some dependencies required by Azure Functions and Durable Functions.

1. Open the `requirements.txt` in the editor and change its content to the following code:

```
azure-functions
azure-functions-durable
```

2. Open the editor's integrated terminal in the current folder (Ctrl+Shift+`).

3. In the integrated terminal, activate the virtual environment in the current folder, depending on your operating system:

- Linux
- MacOS
- Windows

```
source .venv/bin/activate
```

4. In the integrated terminal where the virtual environment is activated, use pip to install the packages you defined.

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

Create your functions

A basic Durable Functions app contains three functions:

- *Orchestrator function*: Describes a workflow that orchestrates other functions.
- *Activity function*: It's called by the orchestrator function, performs work, and optionally returns a value.
- *Client function*: It's a regular Azure Function that starts an orchestrator function. This example uses an HTTP triggered function.

Orchestrator function

You use a template to create the durable function code in your project.

1. In the command palette, search for and select `Azure Functions: Create Function...`.
2. Follow the prompts and provide the following information:

PROMPT	VALUE	DESCRIPTION
Select a template for your function	Durable Functions orchestrator	Create a Durable Functions orchestration
Provide a function name	HelloOrchestrator	Name of your durable function

You've added an orchestrator to coordinate activity functions. Open `HelloOrchestrator/_init_.py` to see the orchestrator function. Each call to `context.call_activity` invokes an activity function named `Hello`.

Next, you'll add the referenced `Hello` activity function.

Activity function

1. In the command palette, search for and select `Azure Functions: Create Function...`.
2. Follow the prompts and provide the following information:

PROMPT	VALUE	DESCRIPTION
Select a template for your function	Durable Functions activity	Create an activity function
Provide a function name	Hello	Name of your activity function

You've added the `Hello` activity function that is invoked by the orchestrator. Open `Hello/_init_.py` to see that it takes a name as input and returns a greeting. An activity function is where you'll perform actions such as making a database call or performing a computation.

Finally, you'll add an HTTP triggered function that starts the orchestration.

Client function (HTTP starter)

1. In the command palette, search for and select `Azure Functions: Create Function...`.

2. Follow the prompts and provide the following information:

PROMPT	VALUE	DESCRIPTION
Select a template for your function	Durable Functions HTTP starter	Create an HTTP starter function
Provide a function name	DurableFunctionsHttpStart	Name of your client function
Authorization level	Anonymous	For demo purposes, allow the function to be called without authentication

You've added an HTTP triggered function that starts an orchestration. Open `DurableFunctionsHttpStart/_init_.py` to see that it uses `client.start_new` to start a new orchestration. Then it uses `client.create_check_status_response` to return an HTTP response containing URLs that can be used to monitor and manage the new orchestration.

You now have a Durable Functions app that can be run locally and deployed to Azure.

Test the function locally

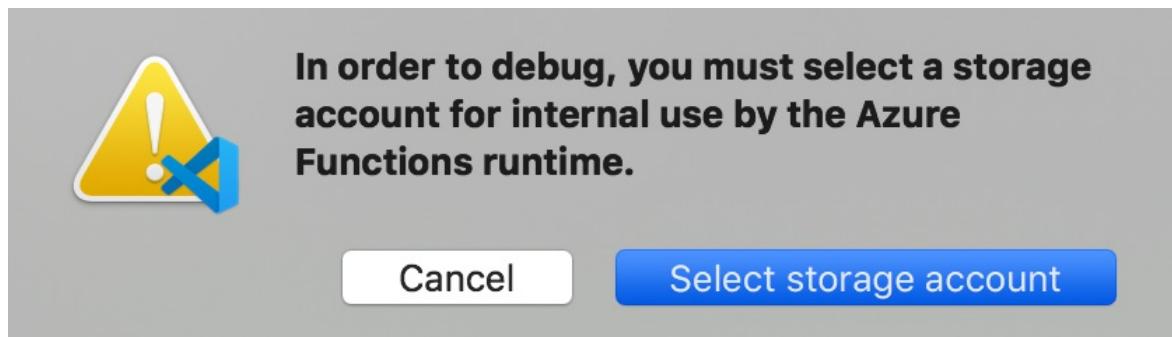
Azure Functions Core Tools lets you run an Azure Functions project on your local development computer. If you don't have it installed, you're prompted to install these tools the first time you start a function from Visual Studio Code.

1. To test your function, set a breakpoint in the `Hello` activity function code (`Hello/_init_.py`). Press F5 or select `Debug: Start Debugging` from the command palette to start the function app project. Output from Core Tools is displayed in the `Terminal` panel.

NOTE

For more information on debugging, see [Durable Functions Diagnostics](#).

2. Durable Functions require an Azure storage account to run. When Visual Studio Code prompts you to select a storage account, select `Select storage account`.



3. Follow the prompts and provide the following information to create a new storage account in Azure:

PROMPT	VALUE	DESCRIPTION
Select subscription	<i>name of your subscription</i>	Select your Azure subscription
Select a storage account	Create a new storage account	
Enter the name of the new storage account	<i>unique name</i>	Name of the storage account to create
Select a resource group	<i>unique name</i>	Name of the resource group to create
Select a location	<i>region</i>	Select a region close to you

4. In the **Terminal** panel, copy the URL endpoint of your HTTP-triggered function.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 2: Task - runFunctionsH ▾ + ⚡ 🗑 ⌂ ×  
Now listening on: http://0.0.0.0:7071  
Application started. Press Ctrl+C to shut down.  
  
Http Functions:  
  
HttpStart: [POST] http://localhost:7071/api/orchestrators/{functionName}
```

5. Use your browser, or a tool like [Postman](#) or [cURL](#), send an HTTP request to the URL endpoint. Replace the last segment with the name of the orchestrator function (`HelloOrchestrator`). The URL must be similar to `http://localhost:7071/api/orchestrators/HelloOrchestrator`.

The response is the initial result from the HTTP function letting you know the durable orchestration has started successfully. It isn't yet the end result of the orchestration. The response includes a few useful URLs. For now, let's query the status of the orchestration.

6. Copy the URL value for `statusQueryGetUri`, paste it in the browser's address bar, and execute the request.
Alternatively, you can also continue to use Postman to issue the GET request.

The request will query the orchestration instance for the status. You must get an eventual response, which shows the instance has completed and includes the outputs or results of the durable function. It looks like:

```
{  
  "name": "HelloOrchestrator",  
  "instanceId": "9a528a9e926f4b46b7d3deaa134b7e8a",  
  "runtimeStatus": "Completed",  
  "input": null,  
  "customStatus": null,  
  "output": [  
    "Hello Tokyo!",  
    "Hello Seattle!",  
    "Hello London!"  
  ],  
  "createdTime": "2020-03-18T21:54:49Z",  
  "lastUpdatedTime": "2020-03-18T21:54:54Z"  
}
```

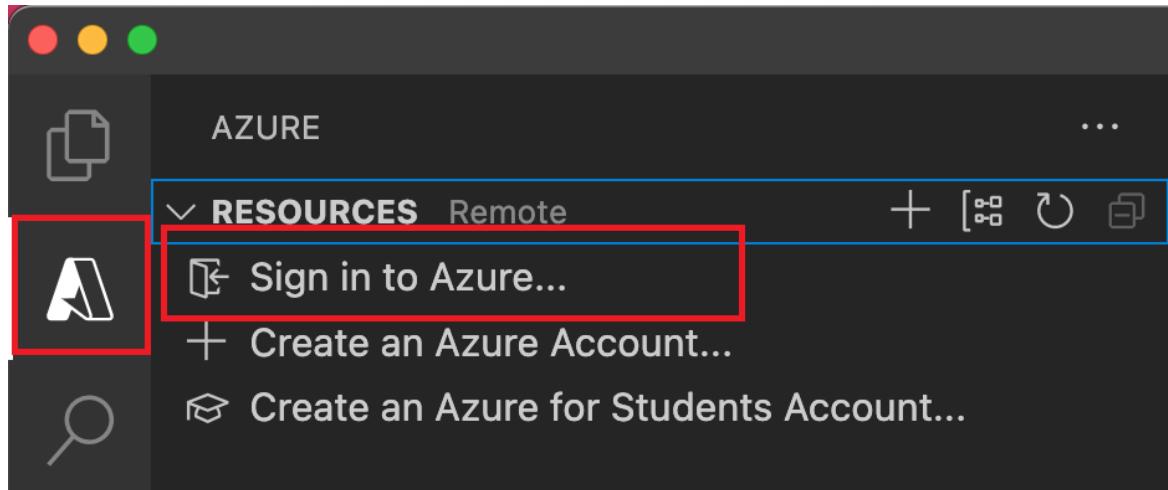
7. To stop debugging, press Shift+F5 in Visual Studio Code.

After you've verified that the function runs correctly on your local computer, it's time to publish the project to Azure.

Sign in to Azure

Before you can publish your app, you must sign in to Azure.

1. If you aren't already signed in, choose the Azure icon in the Activity bar. Then in the Resources area, choose **Sign in to Azure....**



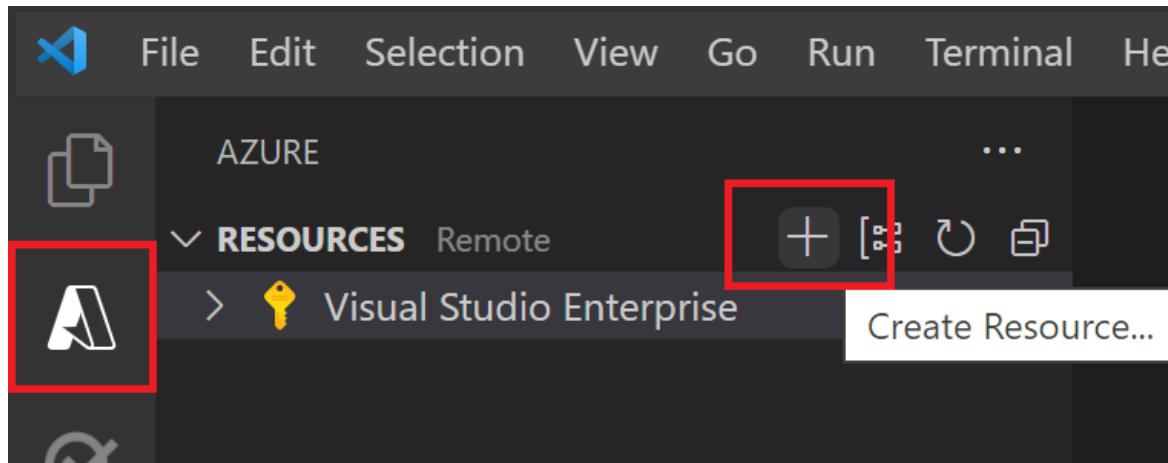
If you're already signed in and can see your existing subscriptions, go to the next section. If you don't yet have an Azure account, choose **Create and Azure Account....** Students can choose **Create and Azure for Students Account....**

2. When prompted in the browser, choose your Azure account and sign in using your Azure account credentials. If you create a new account, you can sign in after your account is created.
3. After you've successfully signed in, you can close the new browser window. The subscriptions that belong to your Azure account are displayed in the sidebar.

Create the function app in Azure

In this section, you create a function app and related resources in your Azure subscription.

1. Choose the Azure icon in the Activity bar. Then in the Resources area, select the + icon and choose the **Create Function App in Azure** option.



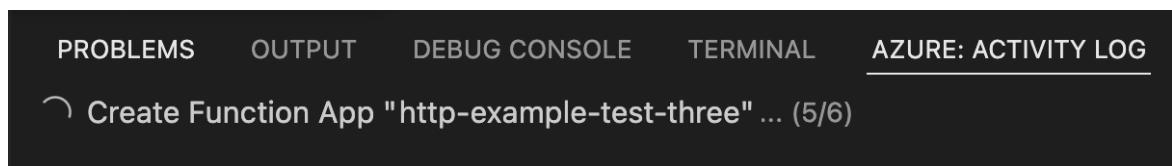
2. Provide the following information at the prompts:

PROMPT

SELECTION

PROMPT	SELECTION
Select subscription	Choose the subscription to use. You won't see this prompt when you have only one subscription visible under Resources .
Enter a globally unique name for the function app	Type a name that is valid in a URL path. The name you type is validated to make sure that it's unique in Azure Functions.
Select a runtime stack	Choose the language version on which you've been running locally.
Select a location for new resources	For better performance, choose a region near you.

The extension shows the status of individual resources as they're being created in Azure in the **Azure: Activity Log** panel.



- When the creation is complete, the following Azure resources are created in your subscription. The resources are named based on your function app name:
 - A [resource group](#), which is a logical container for related resources.
 - A standard [Azure Storage account](#), which maintains state and other information about your projects.
 - A function app, which provides the environment for executing your function code. A function app lets you group functions as a logical unit for easier management, deployment, and sharing of resources within the same hosting plan.
 - An App Service plan, which defines the underlying host for your function app.
 - An Application Insights instance connected to the function app, which tracks usage of your functions in the app.

A notification is displayed after your function app is created and the deployment package is applied.

TIP

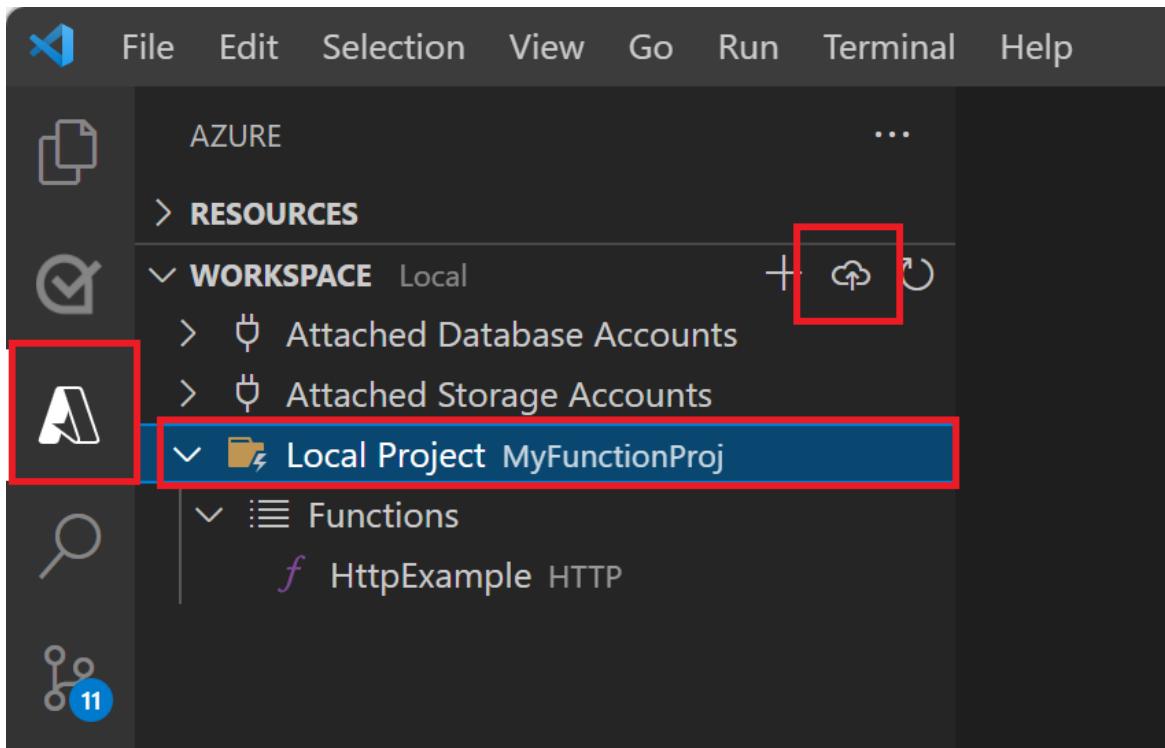
By default, the Azure resources required by your function app are created based on the function app name you provide. By default, they're also created in the same new resource group with the function app. If you want to either customize the names of these resources or reuse existing resources, you need to [publish the project with advanced create options](#) instead.

Deploy the project to Azure

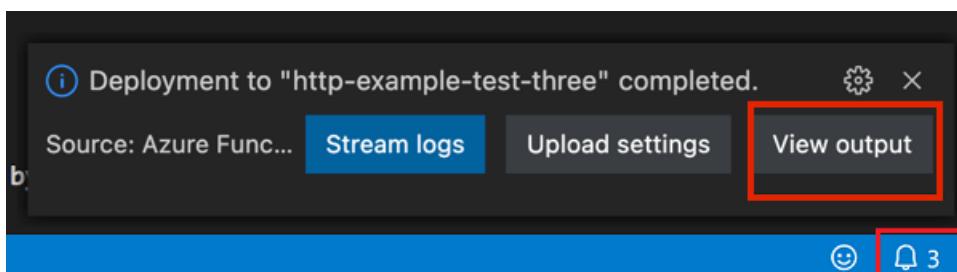
IMPORTANT

Deploying to an existing function app always overwrites the contents of that app in Azure.

- Choose the Azure icon in the Activity bar, then in the **Workspace** area, select your project folder and select the **Deploy...** button.



2. Select **Deploy to Function App...**, choose the function app you just created, and select **Deploy**.
3. After deployment completes, select **View Output** to view the creation and deployment results, including the Azure resources that you created. If you miss the notification, select the bell icon in the lower right corner to see it again.



Test your function in Azure

1. Copy the URL of the HTTP trigger from the **Output** panel. The URL that calls your HTTP-triggered function must be in this format:
`http://<functionappname>.azurewebsites.net/api/orchestrators/HelloOrchestrator`
2. Paste this new URL for the HTTP request in your browser's address bar. You must get the same status response as before when using the published app.

Next steps

You have used Visual Studio Code to create and publish a Python durable function app.

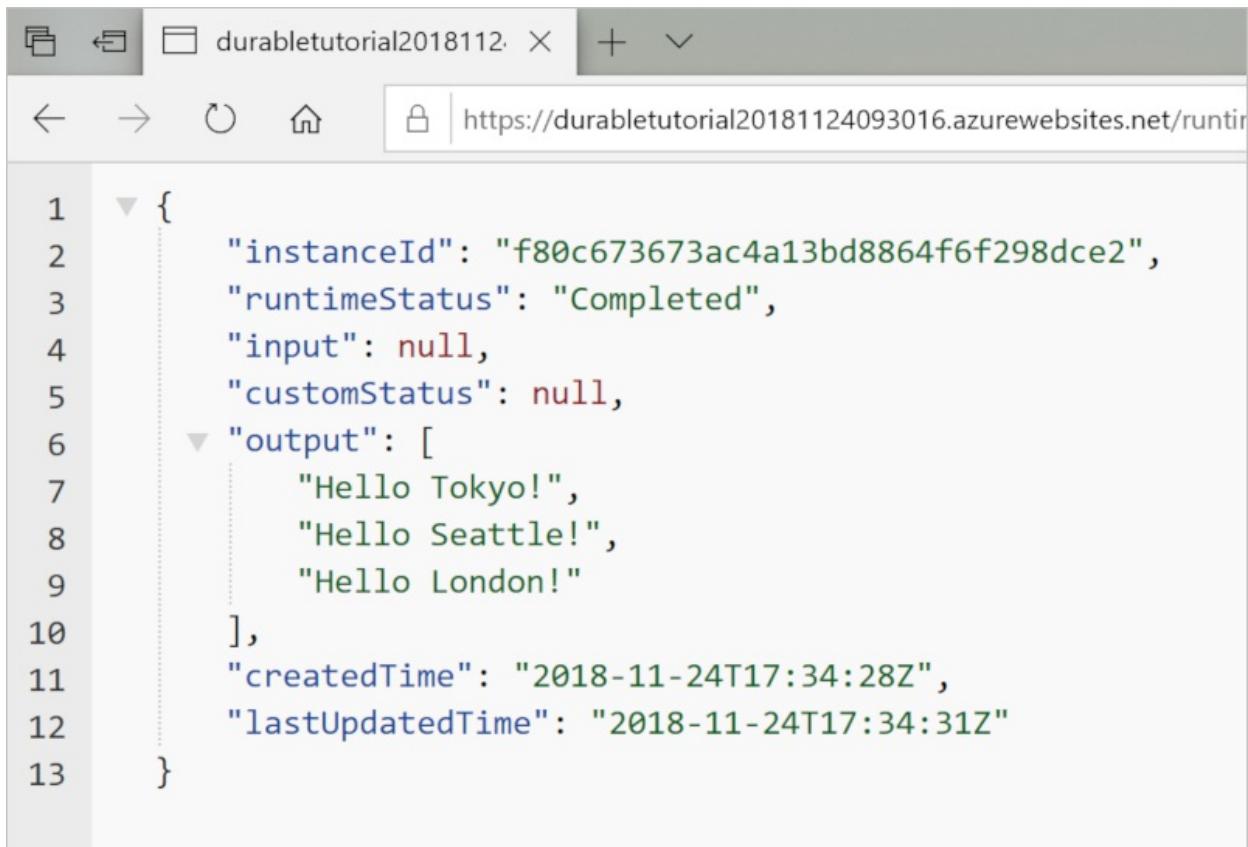
[Learn about common durable function patterns](#)

Create your first durable function in PowerShell

10/5/2022 • 9 minutes to read • [Edit Online](#)

Durable Functions is an extension of [Azure Functions](#) that lets you write stateful functions in a serverless environment. The extension manages state, checkpoints, and restarts for you.

In this article, you learn how to use the Visual Studio Code Azure Functions extension to locally create and test a "hello world" durable function. This function will orchestrate and chain together calls to other functions. You then publish the function code to Azure.



The screenshot shows a browser window with the URL <https://durabletutorial20181124093016.azurewebsites.net/runtime>. The page displays a JSON object representing the state of a durable function instance. The JSON is as follows:

```
1  {
2      "instanceId": "f80c673673ac4a13bd8864f6f298dce2",
3      "runtimeStatus": "Completed",
4      "input": null,
5      "customStatus": null,
6      "output": [
7          "Hello Tokyo!",
8          "Hello Seattle!",
9          "Hello London!"
10     ],
11     "createdTime": "2018-11-24T17:34:28Z",
12     "lastUpdatedTime": "2018-11-24T17:34:31Z"
13 }
```

Prerequisites

To complete this tutorial:

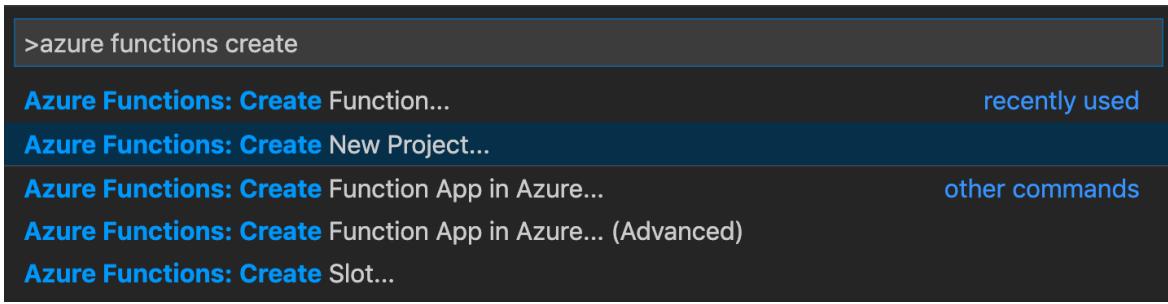
- Install [Visual Studio Code](#).
- Install the [Azure Functions](#) VS Code extension
- Make sure you have the latest version of the [Azure Functions Core Tools](#).
- Durable Functions require an Azure storage account. You need an Azure subscription.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Create your local project

In this section, you use Visual Studio Code to create a local Azure Functions project.

1. In Visual Studio Code, press F1 (or Ctrl/Cmd+Shift+P) to open the command palette. In the command palette, search for and select `Azure Functions: Create New Project...`.



2. Choose an empty folder location for your project and choose **Select**.

3. Following the prompts, provide the following information:

PROMPT	VALUE	DESCRIPTION
Select a language for your function app project	PowerShell	Create a local PowerShell Functions project.
Select a version	Azure Functions v4	You only see this option when the Core Tools aren't already installed. In this case, Core Tools are installed the first time you run the app.
Select a template for your project's first function	Skip for now	
Select how you would like to open your project	Open in current window	Reopens VS Code in the folder you selected.

Visual Studio Code installs the Azure Functions Core Tools, if needed. It also creates a function app project in a folder. This project contains the [host.json](#) and [local.settings.json](#) configuration files.

A package.json file is also created in the root folder.

Configure function app to use PowerShell 7

Open the [local.settings.json](#) file and confirm that a setting named `FUNCTIONS_WORKER_RUNTIME_VERSION` is set to `~7`. If it is missing or set to another value, update the contents of the file.

```
{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "",
    "FUNCTIONS_WORKER_RUNTIME": "powershell",
    "FUNCTIONS_WORKER_RUNTIME_VERSION" : "~7"
  }
}
```

Create your functions

The most basic Durable Functions app contains three functions:

- *Orchestrator function* - describes a workflow that orchestrates other functions.
- *Activity function* - called by the orchestrator function, performs work, and optionally returns a value.
- *Client function* - a regular Azure Function that starts an orchestrator function. This example uses an HTTP triggered function.

Orchestrator function

You use a template to create the durable function code in your project.

1. In the command palette, search for and select `Azure Functions: Create Function...`.

2. Following the prompts, provide the following information:

PROMPT	VALUE	DESCRIPTION
Select a template for your function	Durable Functions orchestrator	Create a Durable Functions orchestration
Provide a function name	HelloOrchestrator	Name of your durable function

You've added an orchestrator to coordinate activity functions. Open `HelloOrchestrator/run.ps1` to see the orchestrator function. Each call to the `Invoke-ActivityFunction` cmdlet invokes an activity function named `Hello`.

Next, you'll add the referenced `Hello` activity function.

Activity function

1. In the command palette, search for and select `Azure Functions: Create Function...`.

2. Following the prompts, provide the following information:

PROMPT	VALUE	DESCRIPTION
Select a template for your function	Durable Functions activity	Create an activity function
Provide a function name	Hello	Name of your activity function

You've added the `Hello` activity function that is invoked by the orchestrator. Open `Hello/run.ps1` to see that it's taking a name as input and returning a greeting. An activity function is where you'll perform actions such as making a database call or performing a computation.

Finally, you'll add an HTTP triggered function that starts the orchestration.

Client function (HTTP starter)

1. In the command palette, search for and select `Azure Functions: Create Function...`.

2. Following the prompts, provide the following information:

PROMPT	VALUE	DESCRIPTION
Select a template for your function	Durable Functions HTTP starter	Create an HTTP starter function
Provide a function name	HttpStart	Name of your activity function
Authorization level	Anonymous	For demo purposes, allow the function to be called without authentication

You've added an HTTP triggered function that starts an orchestration. Open `HttpStart/run.ps1` to see that it uses the `Start-NewOrchestration` cmdlet to start a new orchestration. Then it uses the `New-OrchestrationCheckStatusResponse` cmdlet to return an HTTP response containing URLs that can be used to monitor and manage the new orchestration.

You now have a Durable Functions app that can be run locally and deployed to Azure.

Test the function locally

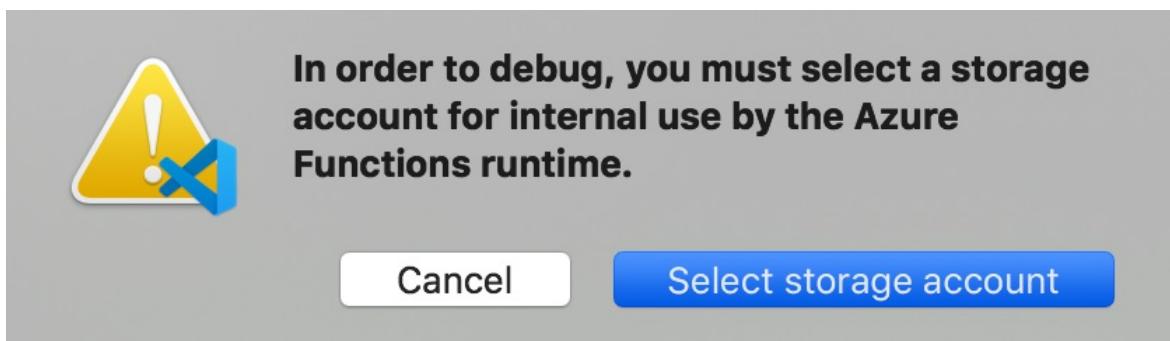
Azure Functions Core Tools lets you run an Azure Functions project on your local development computer. You're prompted to install these tools the first time you start a function app from Visual Studio Code.

1. To test your function, set a breakpoint in the `Hello` activity function code (`Hello/run.ps1`). Press F5 or select `Debug: Start Debugging` from the command palette to start the function app project. Output from Core Tools is displayed in the `Terminal` panel.

NOTE

Refer to the [Durable Functions Diagnostics](#) for more information on debugging.

2. Durable Functions requires an Azure Storage account to run. When VS Code prompts you to select a storage account, choose **Select storage account**.



3. Following the prompts, provide the following information to create a new storage account in Azure.

PROMPT	VALUE	DESCRIPTION
Select subscription	<i>name of your subscription</i>	Select your Azure subscription
Select a storage account	Create a new storage account	
Enter the name of the new storage account	<i>unique name</i>	Name of the storage account to create
Select a resource group	<i>unique name</i>	Name of the resource group to create
Select a location	<i>region</i>	Select a region close to you

4. In the **Terminal** panel, copy the URL endpoint of your HTTP-triggered function.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 2: Task - runFunctionsH ▾ + ⚡ 🗑 ⏷ ⏸ ×  
Now listening on: http://0.0.0.0:7071  
Application started. Press Ctrl+C to shut down.  
  
Http Functions:  
  
HttpStart: [POST] http://localhost:7071/api/orchestrators/{functionName}
```

5. Using your browser, or a tool like [Postman](#) or [cURL](#), send an HTTP POST request to the URL endpoint. Replace the last segment with the name of the orchestrator function (`HelloOrchestrator`). The URL

should be similar to <http://localhost:7071/api/orchestrators/HelloOrchestrator>.

The response is the initial result from the HTTP function letting you know the durable orchestration has started successfully. It is not yet the end result of the orchestration. The response includes a few useful URLs. For now, let's query the status of the orchestration.

6. Copy the URL value for `statusQueryGetUri` and paste it in the browser's address bar and execute the request. Alternatively you can also continue to use Postman to issue the GET request.

The request will query the orchestration instance for the status. You should get an eventual response, which shows us the instance has completed, and includes the outputs or results of the durable function. It looks like:

```
{  
    "name": "HelloOrchestrator",  
    "instanceId": "9a528a9e926f4b46b7d3deaa134b7e8a",  
    "runtimeStatus": "Completed",  
    "input": null,  
    "customStatus": null,  
    "output": [  
        "Hello Tokyo!",  
        "Hello Seattle!",  
        "Hello London!"  
    ],  
    "createdTime": "2020-03-18T21:54:49Z",  
    "lastUpdatedTime": "2020-03-18T21:54:54Z"  
}
```

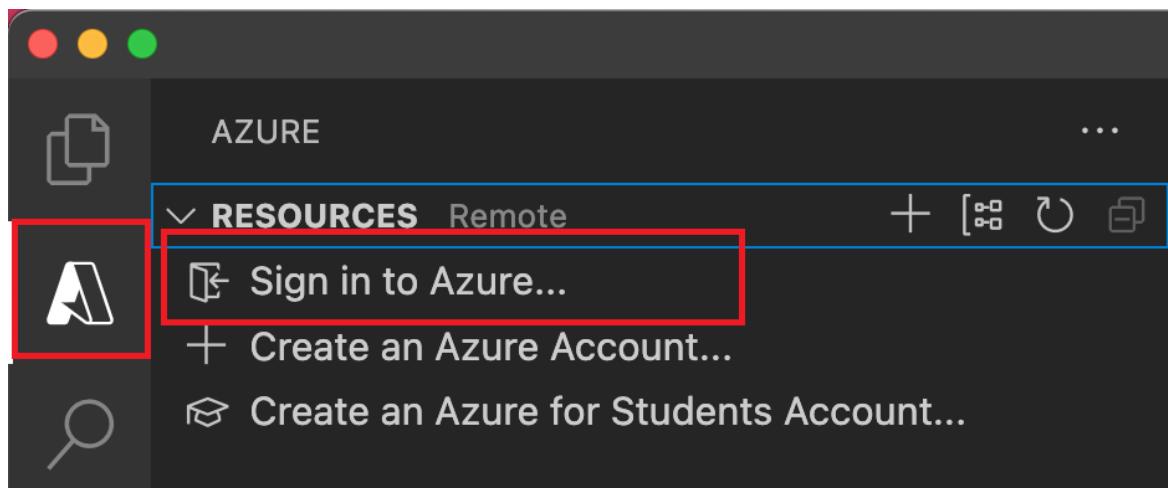
7. To stop debugging, press **Shift + F5** in VS Code.

After you've verified that the function runs correctly on your local computer, it's time to publish the project to Azure.

Sign in to Azure

Before you can publish your app, you must sign in to Azure.

1. If you aren't already signed in, choose the Azure icon in the Activity bar. Then in the Resources area, choose **Sign in to Azure....**



If you're already signed in and can see your existing subscriptions, go to the next section. If you don't yet have an Azure account, choose **Create an Azure Account....** Students can choose **Create an Azure for Students Account....**

2. When prompted in the browser, choose your Azure account and sign in using your Azure account

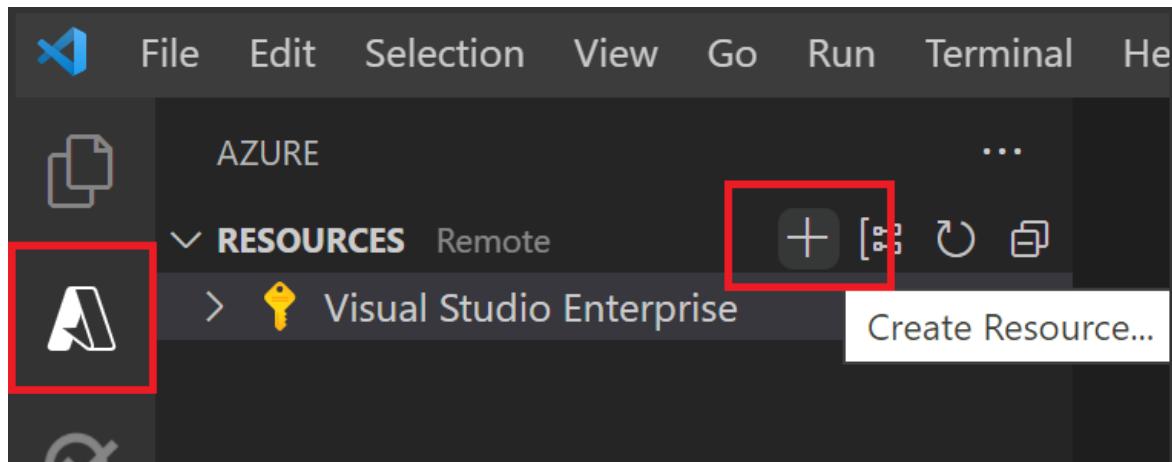
credentials. If you create a new account, you can sign in after your account is created.

3. After you've successfully signed in, you can close the new browser window. The subscriptions that belong to your Azure account are displayed in the sidebar.

Create the function app in Azure

In this section, you create a function app and related resources in your Azure subscription.

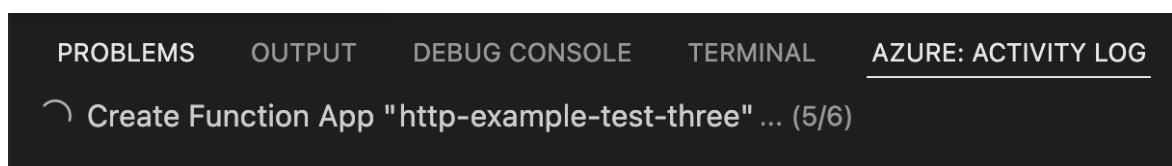
1. Choose the Azure icon in the Activity bar. Then in the **Resources** area, select the + icon and choose the **Create Function App in Azure** option.



2. Provide the following information at the prompts:

PROMPT	SELECTION
Select subscription	Choose the subscription to use. You won't see this prompt when you have only one subscription visible under Resources .
Enter a globally unique name for the function app	Type a name that is valid in a URL path. The name you type is validated to make sure that it's unique in Azure Functions.
Select a runtime stack	Choose the language version on which you've been running locally.
Select a location for new resources	For better performance, choose a region near you.

The extension shows the status of individual resources as they're being created in Azure in the **Azure: Activity Log** panel.



3. When the creation is complete, the following Azure resources are created in your subscription. The resources are named based on your function app name:

- A [resource group](#), which is a logical container for related resources.
- A standard [Azure Storage account](#), which maintains state and other information about your projects.
- A function app, which provides the environment for executing your function code. A function app lets

you group functions as a logical unit for easier management, deployment, and sharing of resources within the same hosting plan.

- An App Service plan, which defines the underlying host for your function app.
- An Application Insights instance connected to the function app, which tracks usage of your functions in the app.

A notification is displayed after your function app is created and the deployment package is applied.

TIP

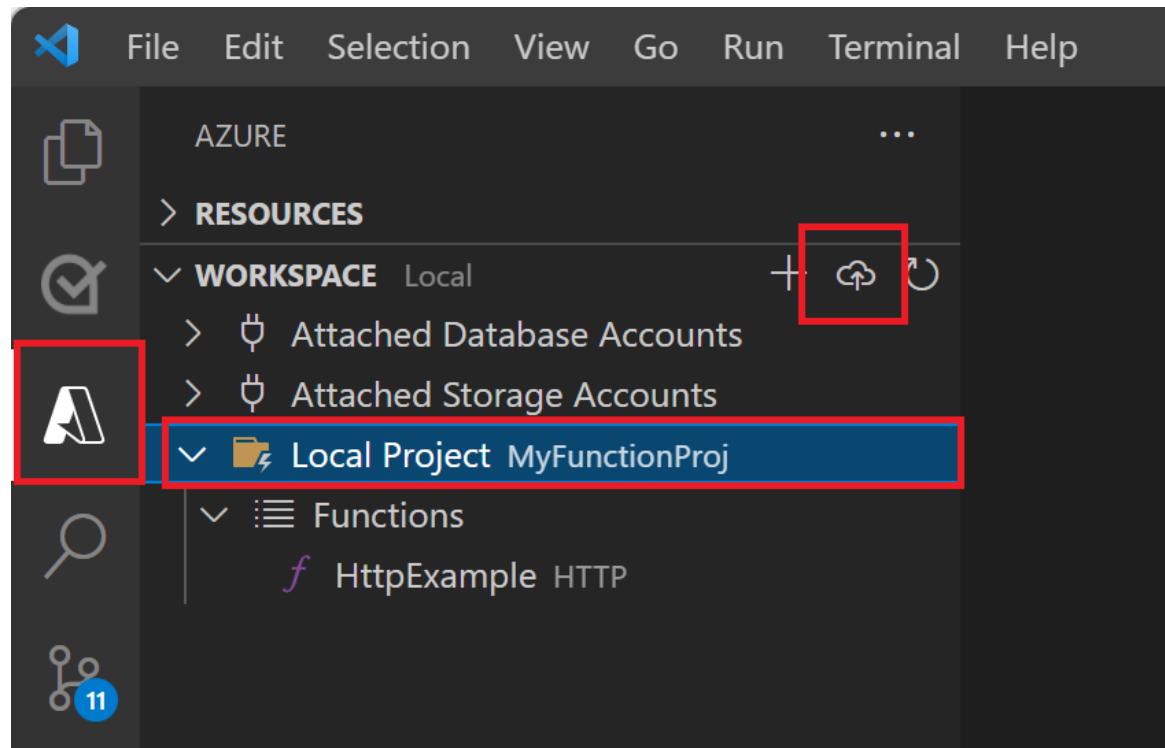
By default, the Azure resources required by your function app are created based on the function app name you provide. By default, they're also created in the same new resource group with the function app. If you want to either customize the names of these resources or reuse existing resources, you need to [publish the project with advanced create options](#) instead.

Deploy the project to Azure

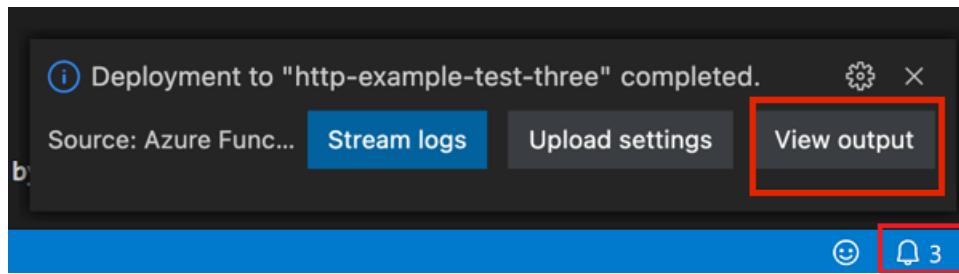
IMPORTANT

Deploying to an existing function app always overwrites the contents of that app in Azure.

1. Choose the Azure icon in the Activity bar, then in the **Workspace** area, select your project folder and select the **Deploy...** button.



2. Select **Deploy to Function App...**, choose the function app you just created, and select **Deploy**.
3. After deployment completes, select **View Output** to view the creation and deployment results, including the Azure resources that you created. If you miss the notification, select the bell icon in the lower right corner to see it again.



Test your function in Azure

1. Copy the URL of the HTTP trigger from the Output panel. The URL that calls your HTTP-triggered function should be in this format:
`http://<functionappname>.azurewebsites.net/api/orchestrators/HelloOrchestrator`
2. Paste this new URL for the HTTP request into your browser's address bar. You should get the same status response as before when using the published app.

Next steps

You have used Visual Studio Code to create and publish a PowerShell durable function app.

[Learn about common durable function patterns](#)

Create your first durable function in Java (Preview)

10/5/2022 • 4 minutes to read • [Edit Online](#)

Durable Functions is an extension of [Azure Functions](#) that lets you write stateful functions in a serverless environment. The extension manages state, checkpoints, and restarts for you.

In this article, you learn how to create and test a "hello world" durable function in your Java project. This function will orchestrate and chain together calls to other functions.

Prerequisites

To complete this tutorial, you need:

- The [Java Developer Kit](#), version 11 or 8.
- [Apache Maven](#), version 3.0 or above.
- Latest version of the [Azure Functions Core Tools](#).
 - For Azure Functions 4.x, Core Tools **v4.0.4590** or newer is required.
- An Azure Storage account, which requires that you have an Azure subscription.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Add required dependencies and plugins

Add the following to your `pom.xml`:

```

<properties>
    <azure.functions.maven.plugin.version>1.18.0</azure.functions.maven.plugin.version>
    <azure.functions.java.library.version>2.0.1</azure.functions.java.library.version>
    <durabletask.azure.functions>1.0.0-beta.1</durabletask.azure.functions>
    <functionAppName>your-unique-app-name</functionAppName>
</properties>

<dependencies>
    <dependency>
        <groupId>com.microsoft.azure.functions</groupId>
        <artifactId>azure-functions-java-library</artifactId>
        <version>${azure.functions.java.library.version}</version>
    </dependency>
    <dependency>
        <groupId>com.microsoft</groupId>
        <artifactId>durabletask-azure-functions</artifactId>
        <version>${durabletask.azure.functions}</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.1</version>
        </plugin>
        <plugin>
            <groupId>com.microsoft.azure</groupId>
            <artifactId>azure-functions-maven-plugin</artifactId>
            <version>${azure.functions.maven.plugin.version}</version>
            <configuration>
                <appName>${functionAppName}</appName>
                <resourceGroup>java-functions-group</resourceGroup>
                <appServicePlanName>java-functions-app-service-plan</appServicePlanName>
                <region>westus</region>
                <runtime>
                    <os>windows</os>
                    <javaVersion>11</javaVersion>
                </runtime>
                <appSettings>
                    <property>
                        <name>FUNCTIONS_EXTENSION_VERSION</name>
                        <value>~4</value>
                    </property>
                </appSettings>
            </configuration>
        </plugin>
        <executions>
            <execution>
                <id>package-functions</id>
                <goals>
                    <goal>package</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
    <plugin>
        <artifactId>maven-clean-plugin</artifactId>
        <version>3.1.0</version>
    </plugin>
</plugins>
</build>

```

Add required JSON files

Add a `host.json` file to your project directory. It should look similar to the following:

```
{  
    "version": "2.0",  
    "logging": {  
        "logLevel": {  
            "DurableTask.AzureStorage": "Warning",  
            "DurableTask.Core": "Warning"  
        }  
    },  
    "extensions": {  
        "durableTask": {  
            "hubName": "JavaTestHub"  
        }  
    },  
    "extensionBundle": {  
        "id": "Microsoft.Azure.Functions.ExtensionBundle.Preview",  
        "version": "[4.*, 5.0.0)"  
    }  
}
```

It's important to note that only the Azure Functions v4 *Preview* bundle currently has the necessary support for Durable Functions for Java.

WARNING

Be aware that the Azure Functions v4 preview bundles do not yet support Cosmos DB bindings for Java function apps. For more information, see [Azure Cosmos DB trigger and bindings reference documentation](#).

Add a `local.settings.json` file to your project directory. You should have the connection string of your Azure Storage account configured for `AzureWebJobsStorage`:

```
{  
    "IsEncrypted": false,  
    "Values": {  
        "AzureWebJobsStorage": "<your storage account connection string>",  
        "FUNCTIONS_WORKER_RUNTIME": "java"  
    }  
}
```

Creating your functions

The most basic Durable Functions app contains three functions:

- *Orchestrator function* - describes a workflow that orchestrates other functions.
- *Activity function* - called by the orchestrator function, performs work, and optionally returns a value.
- *Client function* - a regular Azure Function that starts an orchestrator function. This example uses an HTTP triggered function.

The sample code below shows a simple example of each:

```

import java.util.Optional;

import com.microsoft.azure.functions.*;
import com.microsoft.azure.functions.annotation.*;

import com.microsoft.durabletask.*;
import com.microsoft.durabletask.azurefunctions.*;

public class DurableFunctionsSample {
    /**
     * This HTTP-triggered function starts the orchestration.
     */
    @FunctionName("StartHelloCities")
    public HttpResponseMessage startHelloCities(
        @HttpTrigger(name = "req", methods = {HttpMethod.POST}) HttpRequestMessage<Optional<String>>
req,
        @DurableClientInput(name = "durableContext") DurableClientContext durableContext,
        final ExecutionContext context) {

        DurableTaskClient client = durableContext.getClient();
        String instanceId = client.scheduleNewOrchestrationInstance("HelloCities");
        context.getLogger().info("Created new Java orchestration with instance ID = " + instanceId);
        return durableContext.createCheckStatusResponse(req, instanceId);
    }

    /**
     * This is the orchestrator function, which can schedule activity functions, create durable timers,
     * or wait for external events in a way that's completely fault-tolerant. The
     OrchestratorRunner.loadAndRun()
     * static method is used to take the function input and execute the orchestrator logic.
     */
    @FunctionName("HelloCities")
    public String helloCitiesOrchestrator(@DurableOrchestrationTrigger(name = "runtimeState") String
runtimeState) {
        return OrchestratorRunner.loadAndRun(runtimeState, ctx -> {
            String result = "";
            result += ctx.callActivity("SayHello", "Tokyo", String.class).await() + ", ";
            result += ctx.callActivity("SayHello", "London", String.class).await() + ", ";
            result += ctx.callActivity("SayHello", "Seattle", String.class).await();
            return result;
        });
    }

    /**
     * This is the activity function that gets invoked by the orchestrator function.
     */
    @FunctionName("SayHello")
    public String sayHello(@DurableActivityTrigger(name = "name") String name) {
        return String.format("Hello %s!", name);
    }
}

```

Test the function locally

Azure Functions Core Tools lets you run an Azure Functions project on your local development computer.

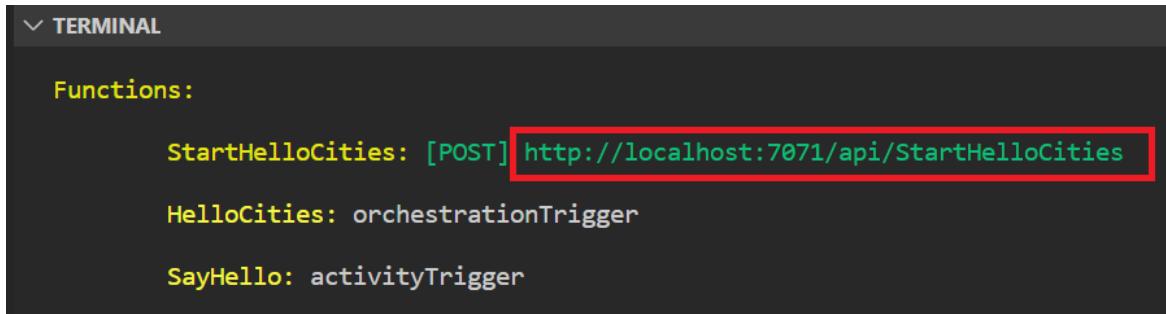
1. If you are using Visual Studio Code, open a new terminal window and run the following commands to build the project:

```
mvn clean package
```

Then run the durable function:

```
mvn azure-functions:run
```

2. In the Terminal panel, copy the URL endpoint of your HTTP-triggered function.



```
▽ TERMINAL

Functions:

StartHelloCities: [POST] http://localhost:7071/api/StartHelloCities
HelloCities: orchestrationTrigger
SayHello: activityTrigger
```

3. Using a tool like [Postman](#) or [cURL](#), send an HTTP POST request to the URL endpoint. You should get a response similar to the following:

```
{
  "id": "d1b33a60-333f-4d6e-9ade-17a7020562a9",
  "purgeHistoryDeleteUri": "http://localhost:7071/runtime/webhooks/durabletask/instances/d1b33a60-333f-4d6e-9ade-17a7020562a9?code=ACCupah_QfGKoFXydcOHH9ffcnYPqjkddSawzRjpp1PQAzFueJ2tDw==",
  "sendEventPostUri": "http://localhost:7071/runtime/webhooks/durabletask/instances/d1b33a60-333f-4d6e-9ade-17a7020562a9/raiseEvent/{eventName}?code=ACCupah_QfGKoFXydcOHH9ffcnYPqjkddSawzRjpp1PQAzFueJ2tDw==",
  "statusQueryGetUri": "http://localhost:7071/runtime/webhooks/durabletask/instances/d1b33a60-333f-4d6e-9ade-17a7020562a9?code=ACCupah_QfGKoFXydcOHH9ffcnYPqjkddSawzRjpp1PQAzFueJ2tDw==",
  "terminatePostUri": "http://localhost:7071/runtime/webhooks/durabletask/instances/d1b33a60-333f-4d6e-9ade-17a7020562a9/terminate?reason={text}&code=ACCupah_QfGKoFXydcOHH9ffcnYPqjkddSawzRjpp1PQAzFueJ2tDw=="}
```

The response is the initial result from the HTTP function letting you know the durable orchestration has started successfully. It is not yet the end result of the orchestration. The response includes a few useful URLs. For now, let's query the status of the orchestration.

4. Copy the URL value for `statusQueryGetUri` and paste it in the browser's address bar and execute the request. Alternatively you can also continue to use Postman or cURL to issue the GET request.

The request will query the orchestration instance for the status. You should get an eventual response, which shows us the instance has completed, and includes the outputs or results of the durable function. It looks like:

```
{
  "name": "HelloCities",
  "instanceId": "d1b33a60-333f-4d6e-9ade-17a7020562a9",
  "runtimeStatus": "Completed",
  "input": null,
  "customStatus": "",
  "output": "Hello Tokyo!, Hello London!, Hello Seattle!",
  "createdTime": "2022-06-15T05:00:02Z",
  "lastUpdatedTime": "2022-06-15T05:00:06Z"
}
```

Function chaining in Durable Functions - Hello sequence sample

10/5/2022 • 9 minutes to read • [Edit Online](#)

Function chaining refers to the pattern of executing a sequence of functions in a particular order. Often the output of one function needs to be applied to the input of another function. This article describes the chaining sequence that you create when you complete the Durable Functions quickstart ([C#](#), [JavaScript](#), [Python](#), [PowerShell](#), or [Java](#)). For more information about Durable Functions, see [Durable Functions overview](#).

Prerequisites

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [Complete the quickstart article](#)
- [Clone or download the samples project from GitHub](#)

The functions

This article explains the following functions in the sample app:

- `E1_HelloSequence` : An [orchestrator function](#) that calls `E1_SayHello` multiple times in a sequence. It stores the outputs from the `E1_SayHello` calls and records the results.
- `E1_SayHello` : An [activity function](#) that prepends a string with "Hello".
- `HttpStart` : An HTTP triggered [durable client](#) function that starts an instance of the orchestrator.

E1_HelloSequence orchestrator function

- [C#](#)
- [JavaScript](#)
- [Python](#)

```
[FunctionName("E1_HelloSequence")]
public static async Task<List<string>> Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    var outputs = new List<string>();

    outputs.Add(await context.CallActivityAsync<string>("E1_SayHello", "Tokyo"));
    outputs.Add(await context.CallActivityAsync<string>("E1_SayHello", "Seattle"));
    outputs.Add(await context.CallActivityAsync<string>("E1_SayHello_DirectInput", "London"));

    // returns ["Hello Tokyo!", "Hello Seattle!", "Hello London!"]
    return outputs;
}
```

All C# orchestration functions must have a parameter of type `DurableOrchestrationContext`, which exists in the `Microsoft.Azure.WebJobs.Extensions.DurableTask` assembly. This context object lets you call other *activity*

functions and pass input parameters using its `CallActivityAsync` method.

The code calls `E1_SayHello` three times in sequence with different parameter values. The return value of each call is added to the `outputs` list, which is returned at the end of the function.

E1_SayHello activity function

- [C#](#)
- [JavaScript](#)
- [Python](#)

```
[FunctionName("E1_SayHello")]
public static string SayHello([ActivityTrigger] IDurableActivityContext context)
{
    string name = context.GetInput<string>();
    return $"Hello {name}!";
}
```

Activities use the `ActivityTrigger` attribute. Use the provided `IDurableActivityContext` to perform activity related actions, such as accessing the input value using `GetInput<T>`.

The implementation of `E1_SayHello` is a relatively trivial string formatting operation.

Instead of binding to an `IDurableActivityContext`, you can bind directly to the type that is passed into the activity function. For example:

```
[FunctionName("E1_SayHello_DirectInput")]
public static string SayHelloDirectInput([ActivityTrigger] string name)
{
    return $"Hello {name}!";
}
```

HttpStart client function

You can start an instance of orchestrator function using a client function. You will use the `HttpStart` HTTP triggered function to start instances of `E1_HelloSequence`.

- [C#](#)
- [JavaScript](#)
- [Python](#)

```

public static class HttpStart
{
    [FunctionName("HttpStart")]
    public static async Task<HttpResponseMessage> Run(
        [HttpTrigger(AuthorizationLevel.Function, methods: "post", Route = "orchestrators/{functionName}")]
    HttpRequestMessage req,
        [DurableClient] IDurableClient starter,
        string functionName,
        ILogger log)
    {
        // Function input comes from the request content.
        object eventData = await req.Content.ReadAsAsync<object>();
        string instanceId = await starter.StartNewAsync(functionName, eventData);

        log.LogInformation($"Started orchestration with ID = '{instanceId}'.");

        return starter.CreateCheckStatusResponse(req, instanceId);
    }
}

```

To interact with orchestrators, the function must include a `DurableClient` input binding. You use the client to start an orchestration. It can also help you return an HTTP response containing URLs for checking the status of the new orchestration.

Run the sample

To execute the `E1_HelloSequence` orchestration, send the following HTTP POST request to the `HttpStart` function.

```
POST http://{host}/orchestrators/E1_HelloSequence
```

NOTE

The previous 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.

For example, if you're running the sample in a function app named "myfunctionapp", replace "{host}" with "myfunctionapp.azurewebsites.net".

The result is an HTTP 202 response, like this (trimmed for brevity):

```

HTTP/1.1 202 Accepted
Content-Length: 719
Content-Type: application/json; charset=utf-8
Location: http://{host}/runtime/webhooks/durabletask/instances/96924899c16d43b08a536de376ac786b?
taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}

(...trimmed...)

```

At this point, the orchestration is queued up and begins to run immediately. The URL in the `Location` header can be used to check the status of the execution.

```
GET http://{host}/runtime/webhooks/durabletask/instances/96924899c16d43b08a536de376ac786b?
taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}
```

The result is the status of the orchestration. It runs and completes quickly, so you see it in the *Completed* state

with a response that looks like this (trimmed for brevity):

```
HTTP/1.1 200 OK
Content-Length: 179
Content-Type: application/json; charset=utf-8

{"runtimeStatus":"Completed","input":null,"output":["Hello Tokyo!","Hello Seattle!","Hello London!"],"createdTime":"2017-06-29T05:24:57Z","lastUpdatedTime":"2017-06-29T05:24:59Z"}
```

As you can see, the `runtimeStatus` of the instance is *Completed* and the `output` contains the JSON-serialized result of the orchestrator function execution.

NOTE

You can implement similar starter logic for other trigger types, like `queueTrigger`, `eventHubTrigger`, or `timerTrigger`.

Look at the function execution logs. The `E1_HelloSequence` function started and completed multiple times due to the replay behavior described in the [orchestration reliability](#) topic. On the other hand, there were only three executions of `E1_SayHello` since those function executions do not get replayed.

Next steps

This sample has demonstrated a simple function-chaining orchestration. The next sample shows how to implement the fan-out/fan-in pattern.

[Run the Fan-out/fan-in sample](#)

Fan-out/fan-in scenario in Durable Functions – Cloud backup example

10/5/2022 • 11 minutes to read • [Edit Online](#)

Fan-out/fan-in refers to the pattern of executing multiple functions concurrently and then performing some aggregation on the results. This article explains a sample that uses [Durable Functions](#) to implement a fan-in/fan-out scenario. The sample is a durable function that backs up all or some of an app's site content into Azure Storage.

Prerequisites

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [Complete the quickstart article](#)
- [Clone or download the samples project from GitHub](#)

Scenario overview

In this sample, the functions upload all files under a specified directory recursively into blob storage. They also count the total number of bytes that were uploaded.

It's possible to write a single function that takes care of everything. The main problem you would run into is **scalability**. A single function execution can only run on a single virtual machine, so the throughput will be limited by the throughput of that single VM. Another problem is **reliability**. If there's a failure midway through, or if the entire process takes more than 5 minutes, the backup could fail in a partially completed state. It would then need to be restarted.

A more robust approach would be to write two regular functions: one would enumerate the files and add the file names to a queue, and another would read from the queue and upload the files to blob storage. This approach is better in terms of throughput and reliability, but it requires you to provision and manage a queue. More importantly, significant complexity is introduced in terms of **state management** and **coordination** if you want to do anything more, like report the total number of bytes uploaded.

A Durable Functions approach gives you all of the mentioned benefits with very low overhead.

The functions

This article explains the following functions in the sample app:

- `E2_BackupSiteContent` : An [orchestrator function](#) that calls `E2_GetFileList` to obtain a list of files to back up, then calls `E2_CopyFileToBlob` to back up each file.
- `E2_GetFileList` : An [activity function](#) that returns a list of files in a directory.
- `E2_CopyFileToBlob` : An activity function that backs up a single file to Azure Blob Storage.

E2_BackupSiteContent orchestrator function

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.

- [C#](#)
- [JavaScript](#)
- [Python](#)

Here is the code that implements the orchestrator function:

```
[FunctionName("E2_BackupSiteContent")]
public static async Task<long> Run(
    [OrchestrationTrigger] IDurableOrchestrationContext backupContext)
{
    string rootDirectory = backupContext.GetInput<string>()?.Trim();
    if (string.IsNullOrEmpty(rootDirectory))
    {
        rootDirectory = Directory.GetParent(typeof(BackupSiteContent).Assembly.Location).FullName;
    }

    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;
}
```

Notice the `await Task.WhenAll(tasks);` line. All the individual calls to the `E2_CopyFileToBlob` function were *not* awaited, which allows 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 virtual machines concurrently, and the Durable Functions extension ensures that the end-to-end execution is resilient to process recycling.

After awaiting from `Task.WhenAll`, 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 `activityTrigger` trigger binding.

`E2_GetFileList` activity function

- [C#](#)
- [JavaScript](#)
- [Python](#)

```
[FunctionName("E2_GetFileList")]
public static string[] GetFileList(
    [ActivityTrigger] string rootDirectory,
    ILogger log)
{
    log.LogInformation($"Searching for files under '{rootDirectory}'...");
    string[] files = Directory.GetFiles(rootDirectory, "*", SearchOption.AllDirectories);
    log.LogInformation($"Found {files.Length} file(s) under {rootDirectory}.");
    return files;
}
```

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. For more information, see [Orchestrator function code constraints](#).

E2_CopyFileToBlob activity function

- [C#](#)
- [JavaScript](#)
- [Python](#)

```
[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, FileAccess.Read, FileShare.Read))
    using (Stream destination = await binder.BindAsync<CloudBlobStream>(
        new BlobAttribute(outputLocation, FileAccess.Write)))
    {
        await source.CopyToAsync(destination);
    }

    return byteCount;
}
```

NOTE

You will need to install the `Microsoft.Azure.WebJobs.Extensions.Storage` NuGet package to run the sample code.

The function uses 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.

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.

NOTE

This is a perfect example of moving I/O operations into an `activityTrigger` function. Not only can the work be distributed across many different machines, 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, on Windows, by sending the following HTTP POST request.

```
POST http://{host}/orchestrators/E2_BackupSiteContent
Content-Type: application/json
Content-Length: 20

"D:\\home\\LogFiles"
```

Alternatively, on a Linux Function App (Python currently only runs on Linux for App Service), you can start the orchestration like so:

```
POST http://{host}/orchestrators/E2_BackupSiteContent
Content-Type: application/json
Content-Length: 20

"/home/site/wwwroot"
```

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 encoded 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}/runtime/webhooks/durabletask/instances/b4e9bdcc435d460f8dc008115ff0a8a9?
taskHub=DurableFunctionsHub&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}/runtime/webhooks/durabletask/instances/b4e9bdcc435d460f8dc008115ff0a8a9?
taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}
```

```
HTTP/1.1 202 Accepted
Content-Length: 148
Content-Type: application/json; charset=utf-8
Location: http://{host}/runtime/webhooks/durabletask/instances/b4e9bdcc435d460f8dc008115ff0a8a9?
taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}

{"runtimeStatus":"Running","input":"D:\\home\\\\LogFiles","output":null,"createdTime":"2019-06-
29T18:50:55Z","lastUpdatedTime":"2019-06-29T18:51:16Z"}
```

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 completion. 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","output":452071,"createdTime":"2019-06-
29T18:50:55Z","lastUpdatedTime":"2019-06-29T18:51:26Z"}
```

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

Next steps

This sample has shown how to implement the fan-out/fan-in pattern. The next sample shows how to implement the monitor pattern using [durable timers](#).

[Run the monitor sample](#)

Monitor scenario in Durable Functions - Weather watcher sample

10/5/2022 • 9 minutes to read • [Edit Online](#)

The monitor pattern refers to a flexible *recurring* process in a workflow - for example, polling until certain conditions are met. This article explains a sample that uses [Durable Functions](#) to implement monitoring.

Prerequisites

- [C#](#)
- [JavaScript](#)
- [Complete the quickstart article](#)
- [Clone or download the samples project from GitHub](#)

Scenario overview

This sample monitors a location's current weather conditions and alerts a user by SMS when the skies are clear. You could use a regular timer-triggered function to check the weather and send alerts. However, one problem with this approach is **lifetime management**. If only one alert should be sent, the monitor needs to disable itself after clear weather is detected. The monitoring pattern can end its own execution, among other benefits:

- Monitors run on intervals, not schedules: a timer trigger *runs* every hour; a monitor *waits* one hour between actions. A monitor's actions will not overlap unless specified, which can be important for long-running tasks.
- Monitors can have dynamic intervals: the wait time can change based on some condition.
- Monitors can terminate when some condition is met or be terminated by another process.
- Monitors can take parameters. The sample shows how the same weather-monitoring process can be applied to any requested location and phone number.
- Monitors are scalable. Because each monitor is an orchestration instance, multiple monitors can be created without having to create new functions or define more code.
- Monitors integrate easily into larger workflows. A monitor can be one section of a more complex orchestration function, or a [sub-orchestration](#).

Configuration

Configuring Twilio integration

This sample involves using the [Twilio](#) service to send SMS messages to a mobile phone. Azure Functions already has support for Twilio via the [Twilio binding](#), and the sample uses that feature.

The first thing you need is a Twilio account. You can create one free at <https://www.twilio.com/try-twilio>. Once you have an account, add the following three **app settings** to your function app.

APP SETTING NAME	VALUE DESCRIPTION
TwilioAccountSid	The SID for your Twilio account
TwilioAuthToken	The Auth token for your Twilio account

APP SETTING NAME	VALUE DESCRIPTION
TwilioPhoneNumber	The phone number associated with your Twilio account. This is used to send SMS messages.

Configuring Weather Underground integration

This sample involves using the Weather Underground API to check current weather conditions for a location.

The first thing you need is a Weather Underground account. You can create one for free at <https://www.wunderground.com/signup>. Once you have an account, you will need to acquire an API key. You can do so by visiting <https://www.wunderground.com/weather/api>, then selecting Key Settings. The Stratus Developer plan is free and sufficient to run this sample.

Once you have an API key, add the following **app setting** to your function app.

APP SETTING NAME	VALUE DESCRIPTION
WeatherUndergroundApiKey	Your Weather Underground API key.

The functions

This article explains the following functions in the sample app:

- `E3_Monitor` : An [orchestrator function](#) that calls `E3_GetIsClear` periodically. It calls `E3_SendGoodWeatherAlert` if `E3_GetIsClear` returns true.
- `E3_GetIsClear` : An [activity function](#) that checks the current weather conditions for a location.
- `E3_SendGoodWeatherAlert` : An activity function that sends an SMS message via Twilio.

E3_Monitor orchestrator function

- [C#](#)
- [JavaScript](#)

```

[FunctionName("E3_Monitor")]
public static async Task Run([OrchestrationTrigger] IDurableOrchestrationContext monitorContext, ILogger log)
{
    MonitorRequest input = monitorContext.GetInput<MonitorRequest>();
    if (!monitorContext.IsReplaying) { log.LogInformation($"Received monitor request. Location: {input?.Location}. Phone: {input?.Phone}."); }

    VerifyRequest(input);

    DateTime endTime = monitorContext.CurrentUtcDateTime.AddHours(6);
    if (!monitorContext.IsReplaying) { log.LogInformation($"Instantiating monitor for {input.Location}. Expires: {endTime}."); }

    while (monitorContext.CurrentUtcDateTime < endTime)
    {
        // Check the weather
        if (!monitorContext.IsReplaying) { log.LogInformation($"Checking current weather conditions for {input.Location} at {monitorContext.CurrentUtcDateTime}."); }

        bool isClear = await monitorContext.CallActivityAsync<bool>("E3_GetIsClear", input.Location);

        if (isClear)
        {
            // It's not raining! Or snowing. Or misting. Tell our user to take advantage of it.
            if (!monitorContext.IsReplaying) { log.LogInformation($"Detected clear weather for {input.Location}. Notifying {input.Phone}."); }

            await monitorContext.CallActivityAsync("E3_SendGoodWeatherAlert", input.Phone);
            break;
        }
        else
        {
            // Wait for the next checkpoint
            var nextCheckpoint = monitorContext.CurrentUtcDateTime.AddMinutes(30);
            if (!monitorContext.IsReplaying) { log.LogInformation($"Next check for {input.Location} at {nextCheckpoint}."); }

            await monitorContext.CreateTimer(nextCheckpoint, CancellationToken.None);
        }
    }

    log.LogInformation($"Monitor expiring.");
}

[Deterministic]
private static void VerifyRequest(MonitorRequest request)
{
    if (request == null)
    {
        throw new ArgumentNullException(nameof(request), "An input object is required.");
    }

    if (request.Location == null)
    {
        throw new ArgumentNullException(nameof(request.Location), "A location input is required.");
    }

    if (string.IsNullOrEmpty(request.Phone))
    {
        throw new ArgumentNullException(nameof(request.Phone), "A phone number input is required.");
    }
}

```

The orchestrator requires a location to monitor and a phone number to send a message to when the weather becomes clear at the location. This data is passed to the orchestrator as a strongly typed `MonitorRequest` object.

This orchestrator function performs the following actions:

1. Gets the **MonitorRequest** consisting of the *location* to monitor and the *phone number* to which it will send an SMS notification.
2. Determines the expiration time of the monitor. The sample uses a hard-coded value for brevity.
3. Calls **E3_IsClear** to determine whether there are clear skies at the requested location.
4. If the weather is clear, calls **E3_SendGoodWeatherAlert** to send an SMS notification to the requested phone number.
5. Creates a durable timer to resume the orchestration at the next polling interval. The sample uses a hard-coded value for brevity.
6. Continues running until the current UTC time passes the monitor's expiration time, or an SMS alert is sent.

Multiple orchestrator instances can run simultaneously by calling the orchestrator function multiple times. The location to monitor and the phone number to send an SMS alert to can be specified. Finally, do note that the orchestrator function is *not* running while waiting for the timer, so you will not get charged for it.

E3_IsClear activity function

As with other samples, the helper activity functions are regular functions that use the `[activityTrigger]` trigger binding. The **E3_IsClear** function gets the current weather conditions using the Weather Underground API and determines whether the sky is clear.

- [C#](#)
- [JavaScript](#)

```
[FunctionName("E3_IsClear")]
public static async Task<bool> GetIsClear([ActivityTrigger] Location location)
{
    var currentConditions = await WeatherUnderground.GetCurrentConditionsAsync(location);
    return currentConditions.Equals(WeatherCondition.Clear);
}
```

E3_SendGoodWeatherAlert activity function

The **E3_SendGoodWeatherAlert** function uses the Twilio binding to send an SMS message notifying the end user that it's a good time for a walk.

- [C#](#)
- [JavaScript](#)

```
[FunctionName("E3_SendGoodWeatherAlert")]
public static void SendGoodWeatherAlert(
    [ActivityTrigger] string phoneNumber,
    ILogger log,
    [TwilioSms(AccountSidSetting = "TwilioAccountSid", AuthTokenSetting = "TwilioAuthToken", From =
"%TwilioPhoneNumber%")]
        out CreateMessageOptions message)
{
    message = new CreateMessageOptions(new PhoneNumber(phoneNumber));
    message.Body = $"The weather's clear outside! Go take a walk!";
}

internal class WeatherUnderground
{
    private static readonly HttpClient httpClient = new HttpClient();
    private static IReadOnlyDictionary<string, WeatherCondition> weatherMapping = new Dictionary<string,
    WeatherCondition>()
    {
        { "Clear", WeatherCondition.Clear }
    }
}
```

```

        { "Clear", WeatherCondition.Clear },
        { "Overcast", WeatherCondition.Clear },
        { "Cloudy", WeatherCondition.Clear },
        { "Clouds", WeatherCondition.Clear },
        { "Drizzle", WeatherCondition.Precipitation },
        { "Hail", WeatherCondition.Precipitation },
        { "Ice", WeatherCondition.Precipitation },
        { "Mist", WeatherCondition.Precipitation },
        { "Precipitation", WeatherCondition.Precipitation },
        { "Rain", WeatherCondition.Precipitation },
        { "Showers", WeatherCondition.Precipitation },
        { "Snow", WeatherCondition.Precipitation },
        { "Spray", WeatherCondition.Precipitation },
        { "Squall", WeatherCondition.Precipitation },
        { "Thunderstorm", WeatherCondition.Precipitation },
    };
}

internal static async Task<WeatherCondition> GetCurrentConditionsAsync(Location location)
{
    var apiKey = Environment.GetEnvironmentVariable("WeatherUndergroundApiKey");
    if (string.IsNullOrEmpty(apiKey))
    {
        throw new InvalidOperationException("The WeatherUndergroundApiKey environment variable was not set.");
    }

    var callString = string.Format("http://api.wunderground.com/api/{0}/conditions/q/{1}/{2}.json",
        apiKey, location.State, location.City);
    var response = await httpClient.GetAsync(callString);
    var conditions = await response.Content.ReadAsAsync< JObject>();

    JToken currentObservation;
    if (!conditions.TryGetValue("current_observation", out currentObservation))
    {
        JToken error = conditions.SelectToken("response.error");

        if (error != null)
        {
            throw new InvalidOperationException($"API returned an error: {error}.");
        }
        else
        {
            throw new ArgumentException("Could not find weather for this location. Try being more specific.");
        }
    }

    return MapToWeatherCondition((string)(currentObservation as JObject).GetValue("weather"));
}

private static WeatherCondition MapToWeatherCondition(string weather)
{
    foreach (var pair in weatherMapping)
    {
        if (weather.Contains(pair.Key))
        {
            return pair.Value;
        }
    }

    return WeatherCondition.Other;
}
}

```

NOTE

You will need to install the `Microsoft.Azure.WebJobs.Extensions.Twilio` Nuget package to run the sample code.

Run the sample

Using the HTTP-triggered functions included in the sample, you can start the orchestration by sending the following HTTP POST request:

```
POST https://{{host}}/orchestrators/E3_Monitor
Content-Length: 77
Content-Type: application/json

{ "location": { "city": "Redmond", "state": "WA" }, "phone": "+1425XXXXXXX" }
```

```
HTTP/1.1 202 Accepted
Content-Type: application/json; charset=utf-8
Location: https://{{host}}/runtime/webhooks/durabletask/instances/f6893f25acf64df2ab53a35c09d52635?
taskHub=SampleHubVS&connection=Storage&code={{SystemKey}}
RetryAfter: 10

{"id": "f6893f25acf64df2ab53a35c09d52635", "statusQueryGetUri":
"https://{{host}}/runtime/webhooks/durabletask/instances/f6893f25acf64df2ab53a35c09d52635?
taskHub=SampleHubVS&connection=Storage&code={{systemKey}}", "sendEventPostUri":
"https://{{host}}/runtime/webhooks/durabletask/instances/f6893f25acf64df2ab53a35c09d52635/raiseEvent/{eventName}?
taskHub=SampleHubVS&connection=Storage&code={{systemKey}}", "terminatePostUri":
"https://{{host}}/runtime/webhooks/durabletask/instances/f6893f25acf64df2ab53a35c09d52635/terminate?reason=
{text}&taskHub=SampleHubVS&connection=Storage&code={{systemKey}}"}
```

The `E3_Monitor` instance starts and queries the current weather conditions for the requested location. If the weather is clear, it calls an activity function to send an alert; otherwise, it sets a timer. When the timer expires, the orchestration will resume.

You can see the orchestration's activity by looking at the function logs in the Azure Functions portal.

```
2018-03-01T01:14:41.649 Function started (Id=2d5fcadf-275b-4226-a174-f9f943c90cd1)
2018-03-01T01:14:42.741 Started orchestration with ID = '1608200bb2ce4b7face5fc3b8e674f2e'.
2018-03-01T01:14:42.780 Function completed (Success, Id=2d5fcadf-275b-4226-a174-f9f943c90cd1,
Duration=1111ms)
2018-03-01T01:14:52.765 Function started (Id=b1b7eb4a-96d3-4f11-a0ff-893e08dd4cfb)
2018-03-01T01:14:52.890 Received monitor request. Location: Redmond, WA. Phone: +1425XXXXXXX.
2018-03-01T01:14:52.895 Instantiating monitor for Redmond, WA. Expires: 3/1/2018 7:14:52 AM.
2018-03-01T01:14:52.909 Checking current weather conditions for Redmond, WA at 3/1/2018 1:14:52 AM.
2018-03-01T01:14:52.954 Function completed (Success, Id=b1b7eb4a-96d3-4f11-a0ff-893e08dd4cfb,
Duration=189ms)
2018-03-01T01:14:53.226 Function started (Id=80a4cb26-c4be-46ba-85c8-ea0c6d07d859)
2018-03-01T01:14:53.808 Function completed (Success, Id=80a4cb26-c4be-46ba-85c8-ea0c6d07d859,
Duration=582ms)
2018-03-01T01:14:53.967 Function started (Id=561d0c78-ee6e-46cb-b6db-39ef639c9a2c)
2018-03-01T01:14:53.996 Next check for Redmond, WA at 3/1/2018 1:44:53 AM.
2018-03-01T01:14:54.030 Function completed (Success, Id=561d0c78-ee6e-46cb-b6db-39ef639c9a2c, Duration=62ms)
```

The orchestration completes once its timeout is reached or clear skies are detected. You can also use the `terminate` API inside another function or invoke the `terminatePostUri` HTTP POST webhook referenced in the 202 response above. To use the webhook, replace `{text}` with the reason for the early termination. The HTTP POST URL will look roughly as follows:

```
POST https://{{host}}/runtime/webhooks/durabletask/instances/f6893f25acf64df2ab53a35c09d52635/terminate?  
reason=Because&taskHub=SampleHubVS&connection=Storage&code={{systemKey}}
```

Next steps

This sample has demonstrated how to use Durable Functions to monitor an external source's status using [durable timers](#) and conditional logic. The next sample shows how to use external events and [durable timers](#) to handle human interaction.

[Run the human interaction sample](#)

Monitor scenario in Durable Functions - GitHub Issue monitoring sample

10/5/2022 • 6 minutes to read • [Edit Online](#)

The monitor pattern refers to a flexible recurring process in a workflow - for example, polling until certain conditions are met. This article explains a sample that uses Durable Functions to implement monitoring.

Prerequisites

- [Complete the quickstart article](#)
- [Clone or download the samples project from GitHub](#)

Scenario overview

This sample monitors the count of issues in a GitHub repo and alerts the user if there are more than 3 open issues. You could use a regular timer-triggered function to request the opened issue counts at regular intervals. However, one problem with this approach is **lifetime management**. If only one alert should be sent, the monitor needs to disable itself after 3 or more issues are detected. The monitoring pattern can end its own execution, among other benefits:

- Monitors run on intervals, not schedules: a timer trigger *runs* every hour; a monitor *waits* one hour between actions. A monitor's actions will not overlap unless specified, which can be important for long-running tasks.
- Monitors can have dynamic intervals: the wait time can change based on some condition.
- Monitors can terminate when some condition is met or be terminated by another process.
- Monitors can take parameters. The sample shows how the same repo-monitoring process can be applied to any requested public GitHub repo and phone number.
- Monitors are scalable. Because each monitor is an orchestration instance, multiple monitors can be created without having to create new functions or define more code.
- Monitors integrate easily into larger workflows. A monitor can be one section of a more complex orchestration function, or a [sub-orchestration](#).

Configuration

Configuring Twilio integration

This sample involves using the [Twilio](#) service to send SMS messages to a mobile phone. Azure Functions already has support for Twilio via the [Twilio binding](#), and the sample uses that feature.

The first thing you need is a Twilio account. You can create one free at <https://www.twilio.com/try-twilio>. Once you have an account, add the following three **app settings** to your function app.

APP SETTING NAME	VALUE DESCRIPTION
TwilioAccountSid	The SID for your Twilio account
TwilioAuthToken	The Auth token for your Twilio account
TwilioPhoneNumber	The phone number associated with your Twilio account. This is used to send SMS messages.

The functions

This article explains the following functions in the sample app:

- `E3_Monitor` : An [orchestrator function](#) that calls `E3_TooManyOpenIssues` periodically. It calls `E3_SendAlert` if the return value of `E3_TooManyOpenIssues` is `True`.
- `E3_TooManyOpenIssues` : An [activity function](#) that checks if a repository has too many open issues. For demoing purposes, we consider having more than 3 open issues to be too many.
- `E3_SendAlert` : An activity function that sends an SMS message via Twilio.

E3_Monitor orchestrator function

The `E3_Monitor` function uses the standard `function.json` for orchestrator functions.

```
{  
    "scriptFile": "__init__.py",  
    "bindings": [  
        {  
            "name": "context",  
            "type": "orchestrationTrigger",  
            "direction": "in"  
        }  
    ]  
}
```

Here is the code that implements the function:

```
import azure.durable_functions as df  
from datetime import timedelta  
from typing import Dict  
  
def orchestrator_function(context: df.DurableOrchestrationContext):  
  
    monitoring_request: Dict[str, str] = context.get_input()  
    repo_url: str = monitoring_request["repo"]  
    phone: str = monitoring_request["phone"]  
  
    # Expiration of the repo monitoring  
    expiry_time = context.current_utc_datetime + timedelta(minutes=5)  
    while context.current_utc_datetime < expiry_time:  
        # Count the number of issues in the repo (the GitHub API caps at 30 issues per page)  
        too_many_issues = yield context.call_activity("E3_TooManyOpenIssues", repo_url)  
  
        # If we detect too many issues, we text the provided phone number  
        if too_many_issues:  
            # Extract URLs of GitHub issues, and return them  
            yield context.call_activity("E3_SendAlert", phone)  
            break  
        else:  
  
            # Reporting the number of statuses found  
            status = f"The repository does not have too many issues, for now ..."  
            context.set_custom_status(status)  
  
            # Schedule a new "wake up" signal  
            next_check = context.current_utc_datetime + timedelta(minutes=1)  
            yield context.create_timer(next_check)  
  
    return "Monitor completed!"  
  
main = df.Orchestrator.create(orchestrator_function)
```

This orchestrator function performs the following actions:

1. Gets the *repo* to monitor and the *phone number* to which it will send an SMS notification.
2. Determines the expiration time of the monitor. The sample uses a hard-coded value for brevity.
3. Calls **E3_TooManyOpenIssues** to determine whether there are too many open issues at the requested repo.
4. If there are too many issues, calls **E3_SendAlert** to send an SMS notification to the requested phone number.
5. Creates a durable timer to resume the orchestration at the next polling interval. The sample uses a hard-coded value for brevity.
6. Continues running until the current UTC time passes the monitor's expiration time, or an SMS alert is sent.

Multiple orchestrator instances can run simultaneously by calling the orchestrator function multiple times. The repo to monitor and the phone number to send an SMS alert to can be specified. Finally, do note that the orchestrator function is *not* running while waiting for the timer, so you will not get charged for it.

E3_TooManyOpenIssues activity function

As with other samples, the helper activity functions are regular functions that use the `activityTrigger` trigger binding. The **E3_TooManyOpenIssues** function gets a list of currently open issues on the repo and determines if there are "too many" of them: more than 3 as per our sample.

The *function.json* is defined as follows:

```
{
  "scriptFile": "__init__.py",
  "bindings": [
    {
      "name": "repoID",
      "type": "activityTrigger",
      "direction": "in"
    }
  ]
}
```

And here is the implementation.

```
import requests
import json

def main(repoID: str) -> str:

    # We use the GitHub API to count the number of open issues in the repo provided
    # Note that the GitHub API only displays at most 30 issues per response, so
    # the maximum number this activity will return is 30. That's enough for demo'ing purposes.
    [user, repo] = repoID.split("/")
    url = f"https://api.github.com/repos/{user}/{repo}/issues?state=open"
    res = requests.get(url)
    if res.status_code != 200:
        error_message = f"Could not find repo {user} under {repo}! API endpoint hit was: {url}"
        raise Exception(error_message)
    issues = json.loads(res.text)
    too_many_issues: bool = len(issues) >= 3
    return too_many_issues
```

E3_SendAlert activity function

The **E3_SendAlert** function uses the Twilio binding to send an SMS message notifying the end user that there are at least 3 open issues awaiting a resolution.

Its *function.json* is simple:

```
{
  "scriptFile": "__init__.py",
  "bindings": [
    {
      "name": "repoID",
      "type": "activityTrigger",
      "direction": "in"
    }
  ]
}
```

And here is the code that sends the SMS message:

```
import json
import random

random.seed(10)

def main(phoneNumber: str, message):
    payload = {
        "body": f"Hey! You may want to check on your repo, there are too many open issues",
        "to": phoneNumber
    }

    message.set(json.dumps(payload))
    return "Message sent!"
```

Run the sample

You will need a [GitHub](#) account. With it, create a temporary public repository that you can open issues to.

Using the HTTP-triggered functions included in the sample, you can start the orchestration by sending the following HTTP POST request:

```
POST https://{{host}}/orchestrators/E3_Monitor
Content-Length: 77
Content-Type: application/json

{ "repo": "<your GitHub handle>/<a new GitHub repo under your user>", "phone": "+1425XXXXXX" }
```

For example, if your GitHub username is `foo` and your repository is `bar` then your value for `"repo"` should be `"foo/bar"`.

```
HTTP/1.1 202 Accepted
Content-Type: application/json; charset=utf-8
Location: https://{{host}}/runtime/webhooks/durabletask/instances/f6893f25acf64df2ab53a35c09d52635?
taskHub=SampleHubVS&connection=Storage&code={SystemKey}
RetryAfter: 10

{"id": "f6893f25acf64df2ab53a35c09d52635", "statusQueryGetUri":
"https://{{host}}/runtime/webhooks/durabletask/instances/f6893f25acf64df2ab53a35c09d52635?
taskHub=SampleHubVS&connection=Storage&code={systemKey}", "sendEventPostUri":
"https://{{host}}/runtime/webhooks/durabletask/instances/f6893f25acf64df2ab53a35c09d52635/raiseEvent/{eventName}?taskHub=SampleHubVS&connection=Storage&code={systemKey}", "terminatePostUri":
"https://{{host}}/runtime/webhooks/durabletask/instances/f6893f25acf64df2ab53a35c09d52635/terminate?reason=
{text}&taskHub=SampleHubVS&connection=Storage&code={systemKey}"}
```

The `E3_Monitor` instance starts and queries the provided repo for open issues. If at least 3 open issues are found, it calls an activity function to send an alert; otherwise, it sets a timer. When the timer expires, the

orchestration will resume.

You can see the orchestration's activity by looking at the function logs in the Azure Functions portal.

```
[2020-12-04T18:24:30.007Z] Executing 'Functions.HttpStart' (Reason='This function was programmatically called via the host APIs.', Id=93772f6b-f4f0-405a-9d7b-be9eb7a38aa6)
[2020-12-04T18:24:30.769Z] Executing 'Functions.E3_Monitor' (Reason='(null)', Id=058e656e-bcb1-418c-95b3-49afcd07bd08)
[2020-12-04T18:24:30.847Z] Started orchestration with ID = '788420bb31754c50acbbc46e12ef4f9c'.
[2020-12-04T18:24:30.932Z] Executed 'Functions.E3_Monitor' (Succeeded, Id=058e656e-bcb1-418c-95b3-49afcd07bd08, Duration=174ms)
[2020-12-04T18:24:30.955Z] Executed 'Functions.HttpStart' (Succeeded, Id=93772f6b-f4f0-405a-9d7b-be9eb7a38aa6, Duration=1028ms)
[2020-12-04T18:24:31.229Z] Executing 'Functions.E3_TooManyOpenIssues' (Reason='(null)', Id=6fd5be5e-7f26-4b0b-98df-c3ac39125da3)
[2020-12-04T18:24:31.772Z] Executed 'Functions.E3_TooManyOpenIssues' (Succeeded, Id=6fd5be5e-7f26-4b0b-98df-c3ac39125da3, Duration=555ms)
[2020-12-04T18:24:40.754Z] Executing 'Functions.E3_Monitor' (Reason='(null)', Id=23915e4c-ddbf-46f9-b3f0-53289ed66082)
[2020-12-04T18:24:40.789Z] Executed 'Functions.E3_Monitor' (Succeeded, Id=23915e4c-ddbf-46f9-b3f0-53289ed66082, Duration=38ms)
(...trimmed...)
```

The orchestration will complete once its timeout is reached or more than 3 open issues are detected. You can also use the `terminate` API inside another function or invoke the `terminatePostUri` HTTP POST webhook referenced in the 202 response above. To use the webhook, replace `{text}` with the reason for the early termination. The HTTP POST URL will look roughly as follows:

```
POST https://{{host}}/runtime/webhooks/durabletask/instances/f6893f25acf64df2ab53a35c09d52635/terminate?
reason=Because&taskHub=SampleHubVS&connection=Storage&code={{systemKey}}
```

Next steps

This sample has demonstrated how to use Durable Functions to monitor an external source's status using [durable timers](#) and conditional logic. The next sample shows how to use external events and [durable timers](#) to handle human interaction.

[Run the human interaction sample](#)

Human interaction in Durable Functions - Phone verification sample

10/5/2022 • 9 minutes to read • [Edit Online](#)

This sample demonstrates how to build a [Durable Functions](#) orchestration that involves human interaction. Whenever a real person is involved in an automated process, the process must be able to send notifications to the person and receive responses asynchronously. It must also allow for the possibility that the person is unavailable. (This last part is where timeouts become important.)

This sample implements an SMS-based phone verification system. These types of flows are often used when verifying a customer's phone number or for multi-factor authentication (MFA). It is a powerful example because the entire implementation is done using a couple small functions. No external data store, such as a database, is required.

Prerequisites

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [Complete the quickstart article](#)
- [Clone or download the samples project from GitHub](#)

Scenario overview

Phone verification is used to verify that end users of your application are not spammers and that they are who they say they are. Multi-factor authentication is a common use case for protecting user accounts from hackers. The challenge with implementing your own phone verification is that it requires a **stateful interaction** with a human being. An end user is typically provided some code (for example, a 4-digit number) and must respond **in a reasonable amount of time**.

Ordinary Azure Functions are stateless (as are many other cloud endpoints on other platforms), so these types of interactions involve explicitly managing state externally in a database or some other persistent store. In addition, the interaction must be broken up into multiple functions that can be coordinated together. For example, you need at least one function for deciding on a code, persisting it somewhere, and sending it to the user's phone. Additionally, you need at least one other function to receive a response from the user and somehow map it back to the original function call in order to do the code validation. A timeout is also an important aspect to ensure security. It can get fairly complex quickly.

The complexity of this scenario is greatly reduced when you use Durable Functions. As you will see in this sample, an orchestrator function can manage the stateful interaction easily and without involving any external data stores. Because orchestrator functions are *durable*, these interactive flows are also highly reliable.

Configuring Twilio integration

This sample involves using the [Twilio](#) service to send SMS messages to a mobile phone. Azure Functions already has support for Twilio via the [Twilio binding](#), and the sample uses that feature.

The first thing you need is a Twilio account. You can create one free at <https://www.twilio.com/try-twilio>. Once you have an account, add the following three **app settings** to your function app.

APP SETTING NAME	VALUE DESCRIPTION
TwilioAccountSid	The SID for your Twilio account
TwilioAuthToken	The Auth token for your Twilio account
TwilioPhoneNumber	The phone number associated with your Twilio account. This is used to send SMS messages.

The functions

This article walks through the following functions in the sample app:

- `E4_SmsPhoneVerification` : An [orchestrator function](#) that performs the phone verification process, including managing timeouts and retries.
- `E4_SendSmsChallenge` : An [activity function](#) that sends a code via text message.

NOTE

The `HttpStart` function in the [sample app](#) and the [quickstart](#) acts as [Orchestration client](#) which triggers the orchestrator function.

E4_SmsPhoneVerification orchestrator function

- [C#](#)
- [JavaScript](#)
- [Python](#)

```

[FunctionName("E4_SmsPhoneVerification")]
public static async Task<bool> Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    string phoneNumber = context.GetInput<string>();
    if (string.IsNullOrEmpty(phoneNumber))
    {
        throw new ArgumentNullException(
            nameof(phoneNumber),
            "A phone number input is required.");
    }

    int challengeCode = await context.CallActivityAsync<int>(
        "E4_SendSmsChallenge",
        phoneNumber);

    using (var timeoutCts = new CancellationTokenSource())
    {
        // The user has 90 seconds to respond with the code they received in the SMS message.
        DateTime expiration = context.CurrentUtcDateTime.AddSeconds(90);
        Task timeoutTask = context.CreateTimer(expiration, timeoutCts.Token);

        bool authorized = false;
        for (int retryCount = 0; retryCount <= 3; retryCount++)
        {
            Task<int> challengeResponseTask =
                context.WaitForExternalEvent<int>("SmsChallengeResponse");

            Task winner = await Task.WhenAny(challengeResponseTask, timeoutTask);
            if (winner == challengeResponseTask)
            {
                // We got back a response! Compare it to the challenge code.
                if (challengeResponseTask.Result == challengeCode)
                {
                    authorized = true;
                    break;
                }
            }
            else
            {
                // Timeout expired
                break;
            }
        }
    }

    if (!timeoutTask.IsCompleted)
    {
        // All pending timers must be complete or canceled before the function exits.
        timeoutCts.Cancel();
    }

    return authorized;
}
}

```

NOTE

It may not be obvious at first, but this orchestrator does not violate the [deterministic orchestration constraint](#). It is deterministic because the `CurrentUtcDateTime` property is used to calculate the timer expiration time, and it returns the same value on every replay at this point in the orchestrator code. This behavior is important to ensure that the same `winner` results from every repeated call to `Task.WhenAny`.

Once started, this orchestrator function does the following:

1. Gets a phone number to which it will *send* the SMS notification.
2. Calls **E4_SendSmsChallenge** to send an SMS message to the user and returns back the expected 4-digit challenge code.
3. Creates a durable timer that triggers 90 seconds from the current time.
4. In parallel with the timer, waits for an **SmsChallengeResponse** event from the user.

The user receives an SMS message with a four-digit code. They have 90 seconds to send that same four-digit code back to the orchestrator function instance to complete the verification process. If they submit the wrong code, they get an additional three tries to get it right (within the same 90-second window).

WARNING

It's important to [cancel timers](#) if you no longer need them to expire, as in the example above when a challenge response is accepted.

E4_SendSmsChallenge activity function

The **E4_SendSmsChallenge** function uses the Twilio binding to send the SMS message with the four-digit code to the end user.

- [C#](#)
- [JavaScript](#)
- [Python](#)

```
[FunctionName("E4_SendSmsChallenge")]
public static int SendSmsChallenge(
    [ActivityTrigger] string phoneNumber,
    ILogger log,
    [TwilioSms(AccountSidSetting = "TwilioAccountSid", AuthTokenSetting = "TwilioAuthToken", From =
    "%TwilioPhoneNumber%")]
        out CreateMessageOptions message)
{
    // Get a random number generator with a random seed (not time-based)
    var rand = new Random(Guid.NewGuid().GetHashCode());
    int challengeCode = rand.Next(10000);

    log.LogInformation($"Sending verification code {challengeCode} to {phoneNumber}. ");

    message = new CreateMessageOptions(new PhoneNumber(phoneNumber));
    message.Body = $"Your verification code is {challengeCode:0000}";

    return challengeCode;
}
```

NOTE

You must first install the `Microsoft.Azure.WebJobs.Extensions.Twilio` Nuget package for Functions to run the sample code. Don't also install the main [Twilio nuget package](#) because this can cause versioning problems that result in build errors.

Run the sample

Using the HTTP-triggered functions included in the sample, you can start the orchestration by sending the following HTTP POST request:

```
POST http://{host}/orchestrators/E4_SmsPhoneVerification
Content-Length: 14
Content-Type: application/json

"+1425XXXXXX"
```

```
HTTP/1.1 202 Accepted
Content-Length: 695
Content-Type: application/json; charset=utf-8
Location: http://{host}/runtime/webhooks/durabletask/instances/741c65651d4c40cea29acdd5bb47baf1?
taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}

{"id":"741c65651d4c40cea29acdd5bb47baf1","statusQueryGetUri":"http://{host}/runtime/webhooks/durabletask/instances/741c65651d4c40cea29acdd5bb47baf1?taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}","sendEventPostUri":"http://{host}/runtime/webhooks/durabletask/instances/741c65651d4c40cea29acd5bb47baf1/raiseEvent/{eventName}?taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}","terminatePostUri":"http://{host}/runtime/webhooks/durabletask/instances/741c65651d4c40cea29acd5bb47baf1/terminate?reason={text}&taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}"}
```

The orchestrator function receives the supplied phone number and immediately sends it an SMS message with a randomly generated 4-digit verification code — for example, `2168`. The function then waits 90 seconds for a response.

To reply with the code, you can use `RaiseEventAsync` (.NET) or `raiseEvent` (JavaScript) inside another function or invoke the `sendEventUrl` HTTP POST webhook referenced in the 202 response above, replacing `{eventName}` with the name of the event `SmsChallengeResponse`:

```
POST
http://{host}/runtime/webhooks/durabletask/instances/741c65651d4c40cea29acdd5bb47baf1/raiseEvent/SmsChallengeResponse?taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}
Content-Length: 4
Content-Type: application/json

2168
```

If you send this before the timer expires, the orchestration completes and the `output` field is set to `true`, indicating a successful verification.

```
GET http://{host}/runtime/webhooks/durabletask/instances/741c65651d4c40cea29acdd5bb47baf1?
taskHub=DurableFunctionsHub&connection=Storage&code={systemKey}
```

```
HTTP/1.1 200 OK
Content-Length: 144
Content-Type: application/json; charset=utf-8

{"runtimeStatus":"Completed","input":"+1425XXXXXX","output":true,"createdTime":"2017-06-29T19:10:49Z","lastUpdatedTime":"2017-06-29T19:12:23Z"}
```

If you let the timer expire, or if you enter the wrong code four times, you can query for the status and see a `false` orchestration function output, indicating that phone verification failed.

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 145

{"runtimeStatus": "Completed", "input": "+1425XXXXXX", "output": false, "createdTime": "2017-06-29T19:20:49Z", "lastUpdatedTime": "2017-06-29T19:22:23Z"}
```

Next steps

This sample has demonstrated some of the advanced capabilities of Durable Functions, notably `WaitForExternalEvent` and `CreateTimer` APIs. You've seen how these can be combined with `Task.WaitAny` to implement a reliable timeout system, which is often useful for interacting with real people. You can learn more about how to use Durable Functions by reading a series of articles that offer in-depth coverage of specific topics.

[Go to the first article in the series](#)

Durable Functions publishing to Azure Event Grid

10/5/2022 • 5 minutes to read • [Edit Online](#)

This article shows how to set up Durable Functions to publish orchestration lifecycle events (such as created, completed, and failed) to a custom [Azure Event Grid Topic](#).

Following are some scenarios where this feature is useful:

- **DevOps scenarios like blue/green deployments:** You might want to know if any tasks are running before implementing the [side-by-side deployment strategy](#).
- **Advanced monitoring and diagnostics support:** You can keep track of orchestration status information in an external store optimized for queries, such as Azure SQL Database or Azure Cosmos DB.
- **Long-running background activity:** If you use Durable Functions for a long-running background activity, this feature helps you to know the current status.

Prerequisites

- Install [Microsoft.Azure.WebJobs.Extensions.DurableTask](#) in your Durable Functions project.
- Install an [Azure Storage Emulator](#) or use an existing Azure Storage account.
- Install [Azure CLI](#) or use [Azure Cloud Shell](#)

Create a custom Event Grid topic

Create an Event Grid topic for sending events from Durable Functions. The following instructions show how to create a topic by using Azure CLI. You can also create the topic by [using PowerShell](#) or by [using the Azure portal](#).

Create a resource group

Create a resource group with the `az group create` command. Currently, Azure Event Grid doesn't support all regions. For information about which regions are supported, see the [Azure Event Grid overview](#).

```
az group create --name eventResourceGroup --location westus2
```

Create a custom topic

An Event Grid topic provides a user-defined endpoint that you post your event to. Replace `<topic_name>` with a unique name for your topic. The topic name must be unique because it becomes a DNS entry.

```
az eventgrid topic create --name <topic_name> -l westus2 -g eventResourceGroup
```

Get the endpoint and key

Get the endpoint of the topic. Replace `<topic_name>` with the name you chose.

```
az eventgrid topic show --name <topic_name> -g eventResourceGroup --query "endpoint" --output tsv
```

Get the topic key. Replace `<topic_name>` with the name you chose.

```
az eventgrid topic key list --name <topic_name> -g eventResourceGroup --query "key1" --output tsv
```

Now you can send events to the topic.

Configure Event Grid publishing

In your Durable Functions project, find the `host.json` file.

Durable Functions 1.x

Add `eventGridTopicEndpoint` and `eventGridKeySettingName` in a `durableTask` property.

```
{
  "durableTask": {
    "eventGridTopicEndpoint": "https://<topic_name>.westus2-1.eventgrid.azure.net/api/events",
    "eventGridKeySettingName": "EventGridKey"
  }
}
```

Durable Functions 2.x

Add a `notifications` section to the `durableTask` property of the file, replacing `<topic_name>` with the name you chose. If the `durableTask` or `extensions` properties don't exist, create them like this example:

```
{
  "version": "2.0",
  "extensions": {
    "durableTask": {
      "notifications": {
        "eventGrid": {
          "topicEndpoint": "https://<topic_name>.westus2-1.eventgrid.azure.net/api/events",
          "keySettingName": "EventGridKey"
        }
      }
    }
  }
}
```

The possible Azure Event Grid configuration properties can be found in the [host.json documentation](#). After you configure the `host.json` file, your function app sends lifecycle events to the Event Grid topic. This action starts when you run your function app both locally and in Azure.

Set the app setting for the topic key in the Function App and `local.settings.json`. The following JSON is a sample of the `local.settings.json` for local debugging using an Azure Storage emulator. Replace `<topic_key>` with the topic key.

```
{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "UseDevelopmentStorage=true",
    "EventGridKey": "<topic_key>"
  }
}
```

If you're using the [Storage Emulator](#) instead of a real Azure Storage account, make sure it's running. It's a good idea to clear any existing storage data before executing.

If you're using a real Azure Storage account, replace `UseDevelopmentStorage=true` in `local.settings.json` with its

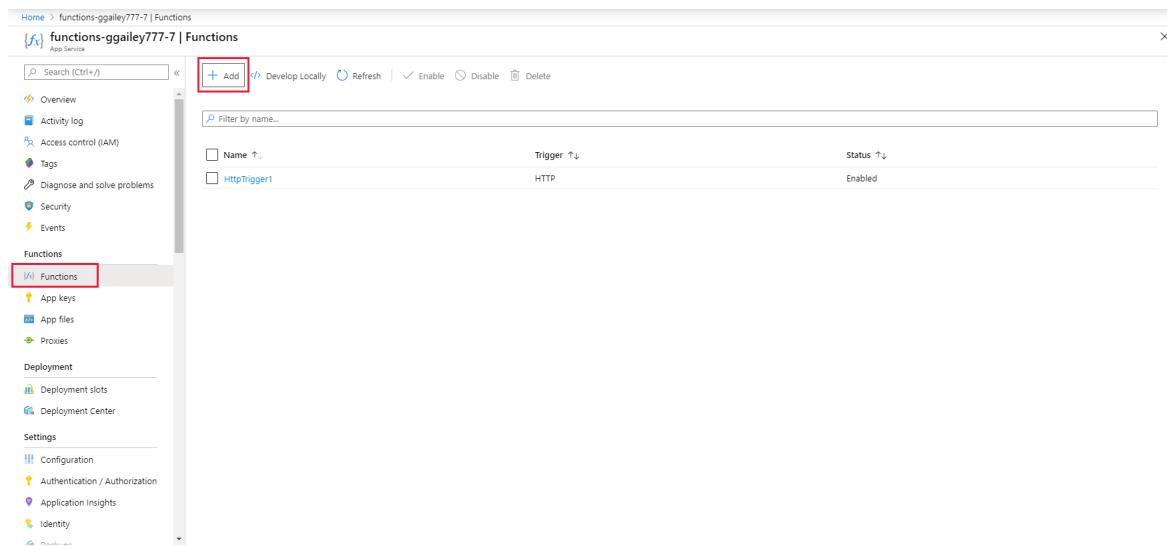
connection string.

Create functions that listen for events

Using the Azure portal, create another function app to listen for events published by your Durable Functions app. It's best to locate it in the same region as the Event Grid topic.

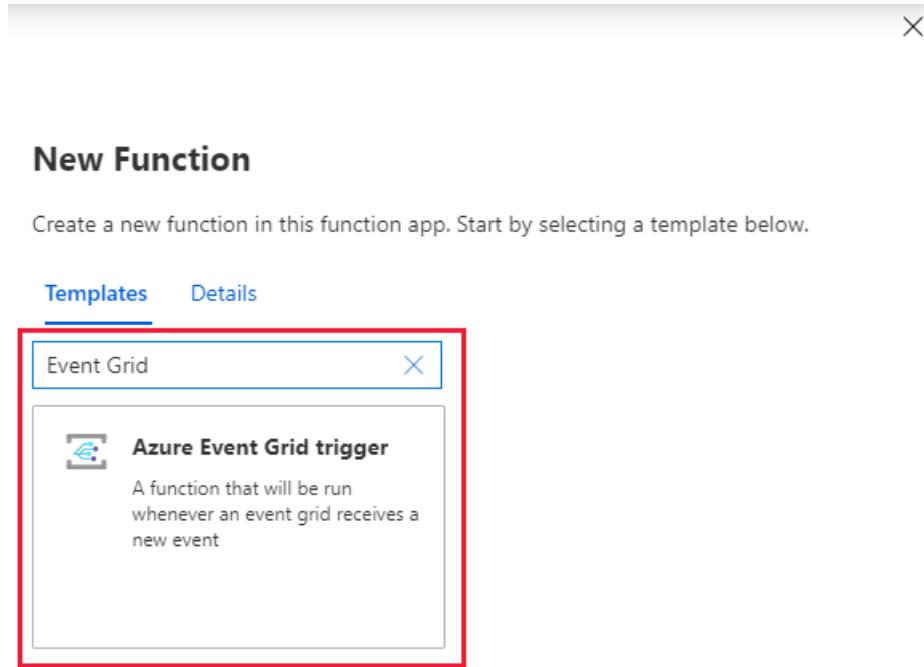
Create an Event Grid trigger function

1. In your function app, select **Functions**, and then select **+ Add**



The screenshot shows the 'functions-gailey777-7 | Functions' blade in the Azure portal. On the left, there's a sidebar with 'Functions' selected. At the top center, there's a search bar and a '+ Add' button, which is highlighted with a red box. Below the search bar, there's a table listing one function: 'HttpTrigger1' with 'HTTP' trigger and 'Enabled' status. There are also filter and sorting options for the table.

2. Search for **Event Grid**, and then select the **Azure Event Grid trigger template**.



The screenshot shows the 'New Function' blade. At the top, there are 'Templates' and 'Details' tabs. The 'Templates' tab is selected. Below it, there's a search bar with 'Event Grid' typed in. Underneath, there's a card for the 'Azure Event Grid trigger' template, which is also highlighted with a red box. The card contains a small icon, the template name, and a brief description: 'A function that will be run whenever an event grid receives a new event'.

3. Name the new trigger, and then select **Create Function**.

New Function

Create a new function in this function app. Start by selecting a template below.

Templates **Details**

New Function*

Create Function

A function with the following code is created:

- C# Script
- JavaScript

```
#r "Newtonsoft.Json"
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;
using Microsoft.Extensions.Logging;

public static void Run(JObject eventGridEvent, ILogger log)
{
    log.LogInformation(eventGridEvent.ToString(Formatting.Indented));
}
```

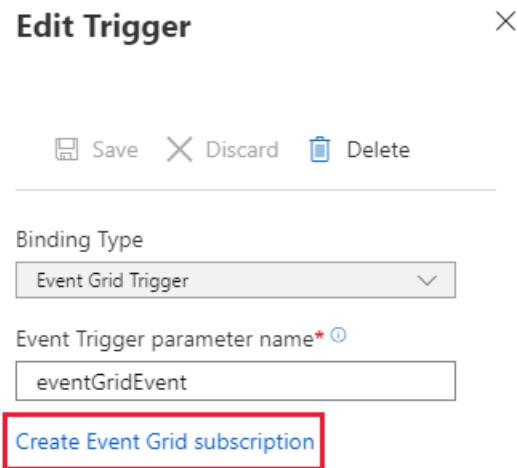
Add an Event Grid subscription

You can now add an Event Grid subscription for the Event Grid topic that you created. For more information, see [Concepts in Azure Event Grid](#).

1. In your new function, select **Integration** and then select **Event Grid Trigger (eventGridEvent)**.

The screenshot shows the Azure Functions blade for a function named 'EventGridTrigger1'. The 'Integration' tab is selected. The 'Trigger' section shows 'Event Grid Trigger (eventGridEvent)' selected. The 'Function' section shows 'EventGridTrigger1'. The 'Inputs' section indicates 'No inputs defined' and has a '+ Add input' button. The 'Outputs' section indicates 'No outputs defined' and has a '+ Add output' button.

2. Select **Create Event Grid Description**.



3. Name your event subscription and select the **Event Grid Topics** topic type.

4. Select the subscription. Then, select the resource group and resource that you created for the Event Grid topic.

5. Select **Create**.

 **Create Event Subscription**
Event Grid

Basic **Filters** **Additional Features**

Event Subscriptions listen for events emitted by the topic resource and send them to the endpoint resource. [Learn more](#)

EVENT SUBSCRIPTION DETAILS

Name	durableSubscription 
Event Schema	Event Grid Schema 

TOPIC DETAILS

Pick a topic resource for which events should be pushed to your destination. [Learn more](#)

Topic Types	Event Grid Topics 
Subscription	Vendor Subscription 
└ Resource Group	NetworkWatcherRG 
└ Resource	Network 

EVENT TYPES

Pick which event types get pushed to your destination. [Learn more](#)

Filter to Event Types  Add Event Type

ENDPOINT DETAILS

Pick an event handler to receive your events. [Learn more](#)

Endpoint Type	 Azure Function (change)
---------------	---

Create

Now you're ready to receive lifecycle events.

Run Durable Functions app to send the events

In the Durable Functions project that you configured earlier, start debugging on your local machine and start an orchestration. The app publishes Durable Functions lifecycle events to Event Grid. Verify that Event Grid triggers the listener function you created by checking its logs in the Azure portal.

```

2019-04-20T09:28:21.041 [Info] Function started (Id=3301c3ef-625f-40ce-ad4c-9ba2916b162d)
2019-04-20T09:28:21.104 [Info] {
    "id": "054fe385-c017-4ce3-b38a-052ac970c39d",
    "subject": "durable/orchestrator/Running",
    "data": {
        "hubName": "DurableFunctionsHub",
        "functionName": "Sample",
        "instanceId": "055d045b1c8a415b94f7671d8df693a6",
        "reason": "",
        "runtimeStatus": "Running"
    },
    "eventType": "orchestratorEvent",
    "eventTime": "2019-04-20T09:28:19.649Z",
    "dataVersion": "1.0",
    "metadataVersion": "1",
    "topic": "/subscriptions/<your_subscription_id>/resourceGroups/eventResourceGroup/providers/Microsoft.EventGrid/topics/durableTopic"
}

2019-04-20T09:28:21.104 [Info] Function completed (Success, Id=3301c3ef-625f-40ce-ad4c-9ba2916b162d, Duration=65ms)
2019-04-20T09:28:37.098 [Info] Function started (Id=36fadea5-198b-4345-bb8e-2837febb89a2)
2019-04-20T09:28:37.098 [Info] {
    "id": "8cf17246-fa9c-4dad-b32a-5a868104f17b",
    "subject": "durable/orchestrator/Completed",
    "data": {
        "hubName": "DurableFunctionsHub",
        "functionName": "Sample",
        "instanceId": "055d045b1c8a415b94f7671d8df693a6",
        "reason": "",
        "runtimeStatus": "Completed"
    },
    "eventType": "orchestratorEvent",
    "eventTime": "2019-04-20T09:28:36.506Z",
    "dataVersion": "1.0",
    "metadataVersion": "1",
    "topic": "/subscriptions/<your_subscription_id>/resourceGroups/eventResourceGroup/providers/Microsoft.EventGrid/topics/durableTopic"
}
2019-04-20T09:28:37.098 [Info] Function completed (Success, Id=36fadea5-198b-4345-bb8e-2837febb89a2, Duration=0ms)

```

Event Schema

The following list explains the lifecycle events schema:

- `id` : Unique identifier for the Event Grid event.
- `subject` : Path to the event subject. `durable/orchestrator/{orchestrationRuntimeStatus}` .
`{orchestrationRuntimeStatus}` will be `Running` , `Completed` , `Failed` , and `Terminated` .
- `data` : Durable Functions Specific Parameters.
 - `hubName` : TaskHub name.
 - `functionName` : Orchestrator function name.
 - `instanceId` : Durable Functions instanceId.
 - `reason` : Additional data associated with the tracking event. For more information, see [Diagnostics in Durable Functions \(Azure Functions\)](#)
 - `runtimeStatus` : Orchestration Runtime Status. Running, Completed, Failed, Canceled.
- `eventType` : "orchestratorEvent"
- `eventTime` : Event time (UTC).

- `dataVersion` : Version of the lifecycle event schema.
- `metadataVersion` : Version of the metadata.
- `topic` : Event grid topic resource.

How to test locally

To test locally, read [Local testing with viewer web app](#). You can also use the *ngrok* utility as shown in [this tutorial](#).

Next steps

[Learn instance management in Durable Functions](#)

[Learn versioning in Durable Functions](#)

Azure CLI Samples

10/5/2022 • 2 minutes to read • [Edit Online](#)

The following table includes links to bash scripts for Azure Functions that use the Azure CLI.

CREATE APP	DESCRIPTION
Create a function app for serverless execution	Create a function app in a Consumption plan.
Create a serverless Python function app	Create a Python function app in a Consumption plan.
Create a function app in a scalable Premium plan	Create a function app in a Premium plan.
Create a function app in a dedicated (App Service) plan	Create a function app in a dedicated App Service plan.
INTEGRATE	DESCRIPTION
Create a function app and connect to a storage account	Create a function app and connect it to a storage account.
Create a function app and connect to an Azure Cosmos DB	Create a function app and connect it to an Azure Cosmos DB.
Create a Python function app and mount a Azure Files share	By mounting a share to your Linux function app, you can leverage existing machine learning models or other data in your functions.
CONTINUOUS DEPLOYMENT	DESCRIPTION
Deploy from GitHub	Create a function app that deploys from a GitHub repository.

Durable Functions types and features

10/5/2022 • 4 minutes to read • [Edit Online](#)

Durable Functions is an extension of [Azure Functions](#). You can use Durable Functions for stateful orchestration of function execution. A durable function app is a solution that's made up of different Azure functions. Functions can play different roles in a durable function orchestration.

There are currently four durable function types in Azure Functions: activity, orchestrator, entity, and client. The rest of this section goes into more details about the types of functions involved in an orchestration.

Orchestrator functions

Orchestrator functions describe how actions are executed and the order in which actions are executed.

Orchestrator functions describe the orchestration in code (C# or JavaScript) as shown in [Durable Functions application patterns](#). An orchestration can have many different types of actions, including [activity functions](#), [sub-orchestrations](#), [waiting for external events](#), [HTTP](#), and [timers](#). Orchestrator functions can also interact with [entity functions](#).

NOTE

Orchestrator functions are written using ordinary code, but there are strict requirements on how to write the code. Specifically, orchestrator function code must be *deterministic*. Failing to follow these determinism requirements can cause orchestrator functions to fail to run correctly. Detailed information on these requirements and how to work around them can be found in the [code constraints](#) topic.

For more detailed information on orchestrator functions and their features, see the [Durable orchestrations](#) article.

Activity functions

Activity functions are the basic unit of work in a durable function orchestration. Activity functions are the functions and tasks that are orchestrated in the process. For example, you might create an orchestrator function to process an order. The tasks involve checking the inventory, charging the customer, and creating a shipment. Each task would be a separate activity function. These activity functions may be executed serially, in parallel, or some combination of both.

Unlike orchestrator functions, activity functions aren't restricted in the type of work you can do in them. Activity functions are frequently used to make network calls or run CPU intensive operations. An activity function can also return data back to the orchestrator function. The Durable Task Framework guarantees that each called activity function will be executed *at least once* during an orchestration's execution.

NOTE

Because activity functions only guarantee *at least once* execution, we recommend you make your activity function logic *idempotent* whenever possible.

Use an [activity trigger](#) to define an activity function. .NET functions receive a `DurableActivityContext` as a parameter. You can also bind the trigger to any other JSON-serializable object to pass in inputs to the function. In JavaScript, you can access an input via the `<activity trigger binding name>` property on the

`context.bindings` object. Activity functions can only have a single value passed to them. To pass multiple values, you must use tuples, arrays, or complex types.

NOTE

You can trigger an activity function only from an orchestrator function.

Entity functions

Entity functions define operations for reading and updating small pieces of state. We often refer to these stateful entities as *durable entities*. Like orchestrator functions, entity functions are functions with a special trigger type, *entity trigger*. They can also be invoked from client functions or from orchestrator functions. Unlike orchestrator functions, entity functions do not have any specific code constraints. Entity functions also manage state explicitly rather than implicitly representing state via control flow.

NOTE

Entity functions and related functionality is only available in Durable Functions 2.0 and above.

For more information about entity functions, see the [Durable Entities](#) article.

Client functions

Orchestrator functions are triggered by an [orchestration trigger binding](#) and entity functions are triggered by an [entity trigger binding](#). Both of these triggers work by reacting to messages that are enqueued into a [task hub](#). The primary way to deliver these messages is by using an [orchestrator client binding](#) or an [entity client binding](#) from within a *client function*. Any non-orchestrator function can be a *client function*. For example, You can trigger the orchestrator from an HTTP-triggered function, an Azure Event Hub triggered function, etc. What makes a function a *client function* is its use of the durable client output binding.

NOTE

Unlike other function types, orchestrator and entity functions cannot be triggered directly using the buttons in the Azure Portal. If you want to test an orchestrator or entity function in the Azure Portal, you must instead run a *client function* that starts an orchestrator or entity function as part of its implementation. For the simplest testing experience, a *manual trigger* function is recommended.

In addition to triggering orchestrator or entity functions, the *durable client* binding can be used to interact with running orchestrations and entities. For example, orchestrations can be queried, terminated, and can have events raised to them. For more information on managing orchestrations and entities, see the [Instance management](#) article.

Next steps

To get started, create your first durable function in [C#](#), [JavaScript](#), [Python](#), [PowerShell](#), or [Java](#).

[Read more about Durable Functions orchestrations](#)

Durable orchestrations

10/5/2022 • 12 minutes to read • [Edit Online](#)

Durable Functions is an extension of [Azure Functions](#). You can use an *orchestrator function* to orchestrate the execution of other Durable functions within a function app. Orchestrator functions have the following characteristics:

- Orchestrator functions define function workflows using procedural code. No declarative schemas or designers are needed.
- Orchestrator functions can call other durable functions synchronously and asynchronously. Output from called functions can be reliably saved to local variables.
- Orchestrator functions are durable and reliable. Execution progress is automatically checkpointed when the function "awaits" or "yields". Local state is never lost when the process recycles or the VM reboots.
- Orchestrator functions can be long-running. The total lifespan of an *orchestration instance* can be seconds, days, months, or never-ending.

This article gives you an overview of orchestrator functions and how they can help you solve various app development challenges. If you are not already familiar with the types of functions available in a Durable Functions app, read the [Durable Function types](#) article first.

Orchestration identity

Each *instance* of an orchestration has an instance identifier (also known as an *instance ID*). By default, each instance ID is an autogenerated GUID. However, instance IDs can also be any user-generated string value. Each orchestration instance ID must be unique within a [task hub](#).

The following are some rules about instance IDs:

- Instance IDs must be between 1 and 256 characters.
- Instance IDs must not start with `@`.
- Instance IDs must not contain `/`, `\`, `#`, or `?` characters.
- Instance IDs must not contain control characters.

NOTE

It is generally recommended to use autogenerated instance IDs whenever possible. User-generated instance IDs are intended for scenarios where there is a one-to-one mapping between an orchestration instance and some external application-specific entity, like a purchase order or a document.

An orchestration's instance ID is a required parameter for most [instance management operations](#). They are also important for diagnostics, such as [searching through orchestration tracking data](#) in Application Insights for troubleshooting or analytics purposes. For this reason, it is recommended to save generated instance IDs to some external location (for example, a database or in application logs) where they can be easily referenced later.

Reliability

Orchestrator functions reliably maintain their execution state by using the [event sourcing](#) design pattern. Instead of directly storing the current state of an orchestration, the Durable Task Framework uses an append-only store to record the full series of actions the function orchestration takes. An append-only store has many benefits compared to "dumping" the full runtime state. Benefits include increased performance, scalability, and

responsiveness. You also get eventual consistency for transactional data and full audit trails and history. The audit trails support reliable compensating actions.

Durable Functions uses event sourcing transparently. Behind the scenes, the `await` (C#) or `yield` (JavaScript/Python) operator in an orchestrator function yields control of the orchestrator thread back to the Durable Task Framework dispatcher. In the case of Java, there is no special language keyword. Instead, calling `.await()` on a task will yield control back to the dispatcher via a custom `Throwable`. The dispatcher then commits any new actions that the orchestrator function scheduled (such as calling one or more child functions or scheduling a durable timer) to storage. The transparent commit action updates the execution history of the orchestration instance by appending all new events into storage, much like an append-only log. Similarly, the commit action creates messages in storage to schedule the actual work. At this point, the orchestrator function can be unloaded from memory. By default, Durable Functions uses Azure Storage as its runtime state store, but other [storage providers are also supported](#).

When an orchestration function is given more work to do (for example, a response message is received or a durable timer expires), the orchestrator wakes up and re-executes the entire function from the start to rebuild the local state. During the replay, if the code tries to call a function (or do any other async work), the Durable Task Framework consults the execution history of the current orchestration. If it finds that the [activity function](#) has already executed and yielded a result, it replays that function's result and the orchestrator code continues to run. Replay continues until the function code is finished or until it has scheduled new async work.

NOTE

In order for the replay pattern to work correctly and reliably, orchestrator function code must be *deterministic*. Non-deterministic orchestrator code may result in runtime errors or other unexpected behavior. For more information about code restrictions for orchestrator functions, see the [orchestrator function code constraints](#) documentation.

NOTE

If an orchestrator function emits log messages, the replay behavior may cause duplicate log messages to be emitted. See the [Logging](#) topic to learn more about why this behavior occurs and how to work around it.

Orchestration history

The event-sourcing behavior of the Durable Task Framework is closely coupled with the orchestrator function code you write. Suppose you have an activity-chaining orchestrator function, like the following orchestrator function:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```

[FunctionName("HelloCities")]
public static async Task<List<string>> Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    var outputs = new List<string>();

    outputs.Add(await context.CallActivityAsync<string>("SayHello", "Tokyo"));
    outputs.Add(await context.CallActivityAsync<string>("SayHello", "Seattle"));
    outputs.Add(await context.CallActivityAsync<string>("SayHello", "London"));

    // returns ["Hello Tokyo!", "Hello Seattle!", "Hello London!"]
    return outputs;
}

```

Whenever an activity function is scheduled, the Durable Task Framework checkpoints the execution state of the function into some durable storage backend (Azure Table storage by default). This state is what's referred to as the *orchestration history*.

History table

Generally speaking, the Durable Task Framework does the following at each checkpoint:

1. Saves execution history into durable storage.
2. Enqueues messages for functions the orchestrator wants to invoke.
3. Enqueues messages for the orchestrator itself — for example, durable timer messages.

Once the checkpoint is complete, the orchestrator function is free to be removed from memory until there is more work for it to do.

NOTE

Azure Storage does not provide any transactional guarantees between saving data into table storage and queues. To handle failures, the [Durable Functions Azure Storage](#) provider uses *eventual consistency* patterns. These patterns ensure that no data is lost if there is a crash or loss of connectivity in the middle of a checkpoint. Alternate storage providers, such as the [Durable Functions MSSQL storage provider](#), may provide stronger consistency guarantees.

Upon completion, the history of the function shown earlier looks something like the following table in Azure Table Storage (abbreviated for illustration purposes):

PARTITIONKEY (INSTANCEID)	EVENTTYPE	TIMESTAMP	INPUT	NAME	RESULT	STATUS
eaee885b	ExecutionStarted	2021-05-05T18:45:28.852Z	null	HelloCities		
eaee885b	OrchestratorStarted	2021-05-05T18:45:32.362Z				
eaee885b	TaskScheduled	2021-05-05T18:45:32.670Z		SayHello		
eaee885b	OrchestratorCompleted	2021-05-05T18:45:32.670Z				

PARTITIONKEY (INSTANCEID)	EVENTTYPE	TIMESTAMP	INPUT	NAME	RESULT	STATUS
eaee885b	TaskCompleted	2021-05-05T18:45:34.201Z			""Hello Tokyo!""	
eaee885b	OrchestratorStarted	2021-05-05T18:45:34.232Z				
eaee885b	TaskScheduled	2021-05-05T18:45:34.435Z		SayHello		
eaee885b	OrchestratorCompleted	2021-05-05T18:45:34.435Z				
eaee885b	TaskCompleted	2021-05-05T18:45:34.763Z			""Hello Seattle!""	
eaee885b	OrchestratorStarted	2021-05-05T18:45:34.857Z				
eaee885b	TaskScheduled	2021-05-05T18:45:34.857Z		SayHello		
eaee885b	OrchestratorCompleted	2021-05-05T18:45:34.857Z				
eaee885b	TaskCompleted	2021-05-05T18:45:34.919Z			""Hello London!""	
eaee885b	OrchestratorStarted	2021-05-05T18:45:35.032Z				
eaee885b	OrchestratorCompleted	2021-05-05T18:45:35.044Z				
eaee885b	ExecutionCompleted	2021-05-05T18:45:35.044Z			["""Hello Tokyo!""", """Hello Seattle!""", """Hello London!"""]"	Completed

A few notes on the column values:

- **PartitionKey:** Contains the instance ID of the orchestration.
- **EventType:** Represents the type of the event. You can find detailed descriptions of all the history event types

[here](#).

- **Timestamp:** The UTC timestamp of the history event.
- **Name:** The name of the function that was invoked.
- **Input:** The JSON-formatted input of the function.
- **Result:** The output of the function; that is, its return value.

WARNING

While it's useful as a debugging tool, don't take any dependency on this table. It may change as the Durable Functions extension evolves.

Every time the function is resumed after waiting for a task to complete, the Durable Task Framework reruns the orchestrator function from scratch. On each rerun, it consults the execution history to determine whether the current async task has completed. If the execution history shows that the task has already completed, the framework replays the output of that task and moves on to the next task. This process continues until the entire execution history has been replayed. Once the current execution history has been replayed, the local variables will have been restored to their previous values.

Features and patterns

The next sections describe the features and patterns of orchestrator functions.

Sub-orchestrations

Orchestrator functions can call activity functions, but also other orchestrator functions. For example, you can build a larger orchestration out of a library of orchestrator functions. Or, you can run multiple instances of an orchestrator function in parallel.

For more information and for examples, see the [Sub-orchestrations](#) article.

Durable timers

Orchestrations can schedule *durable timers* to implement delays or to set up timeout handling on async actions. Use durable timers in orchestrator functions instead of language-native "sleep" APIs.

For more information and for examples, see the [Durable timers](#) article.

External events

Orchestrator functions can wait for external events to update an orchestration instance. This Durable Functions feature often is useful for handling a human interaction or other external callbacks.

For more information and for examples, see the [External events](#) article.

Error handling

Orchestrator functions can use the error-handling features of the programming language. Existing patterns like `try / catch` are supported in orchestration code.

Orchestrator functions can also add retry policies to the activity or sub-orchestrator functions that they call. If an activity or sub-orchestrator function fails with an exception, the specified retry policy can automatically delay and retry the execution up to a specified number of times.

NOTE

If there is an unhandled exception in an orchestrator function, the orchestration instance will complete in a `Failed` state. An orchestration instance cannot be retried once it has failed.

For more information and for examples, see the [Error handling](#) article.

Critical sections (Durable Functions 2.x, currently .NET only)

Orchestration instances are single-threaded so it isn't necessary to worry about race conditions *within* an orchestration. However, race conditions are possible when orchestrations interact with external systems. To mitigate race conditions when interacting with external systems, orchestrator functions can define *critical sections* using a `LockAsync` method in .NET.

The following sample code shows an orchestrator function that defines a critical section. It enters the critical section using the `LockAsync` method. This method requires passing one or more references to a [Durable Entity](#), which durably manages the lock state. Only a single instance of this orchestration can execute the code in the critical section at a time.

```
[FunctionName("Synchronize")]
public static async Task Synchronize(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    var lockId = new EntityId("LockEntity", "MyLockIdentifier");
    using (await context.LockAsync(lockId))
    {
        // critical section - only one orchestration can enter at a time
    }
}
```

The `LockAsync` acquires the durable lock(s) and returns an `IDisposable` that ends the critical section when disposed. This `IDisposable` result can be used together with a `using` block to get a syntactic representation of the critical section. When an orchestrator function enters a critical section, only one instance can execute that block of code. Any other instances that try to enter the critical section will be blocked until the previous instance exits the critical section.

The critical section feature is also useful for coordinating changes to durable entities. For more information about critical sections, see the [Durable entities "Entity coordination"](#) topic.

NOTE

Critical sections are available in Durable Functions 2.0 and above. Currently, only .NET orchestrations implement this feature.

Calling HTTP endpoints (Durable Functions 2.x)

Orchestrator functions aren't permitted to do I/O, as described in [orchestrator function code constraints](#). The typical workaround for this limitation is to wrap any code that needs to do I/O in an activity function.

Orchestrations that interact with external systems frequently use activity functions to make HTTP calls and return the result to the orchestration.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

To simplify this common pattern, orchestrator functions can use the `CallHttpAsync` method to invoke HTTP APIs directly.

```
[FunctionName("CheckSiteAvailable")]
public static async Task CheckSiteAvailable(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    Uri url = context.GetInput<Uri>();

    // Makes an HTTP GET request to the specified endpoint
    DurableHttpResponse response =
        await context.CallHttpAsync(HttpMethod.Get, url);

    if ((int)response.StatusCode == 400)
    {
        // handling of error codes goes here
    }
}
```

In addition to supporting basic request/response patterns, the method supports automatic handling of common async HTTP 202 polling patterns, and also supports authentication with external services using [Managed Identities](#).

For more information and for detailed examples, see the [HTTP features](#) article.

NOTE

Calling HTTP endpoints directly from orchestrator functions is available in Durable Functions 2.0 and above.

Passing multiple parameters

It isn't possible to pass multiple parameters to an activity function directly. The recommendation is to pass in an array of objects or composite objects.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

In .NET you can also use [ValueTuple](#) objects. The following sample is using new features of [ValueTuple](#) added with [C# 7](#):

```

[FunctionName("GetCourseRecommendations")]
public static async Task<object> RunOrchestrator(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    string major = "ComputerScience";
    int universityYear = context.GetInput<int>();

    object courseRecommendations = await context.CallActivityAsync<object>(
        "CourseRecommendations",
        (major, universityYear));
    return courseRecommendations;
}

[FunctionName("CourseRecommendations")]
public static async Task<object> Mapper([ActivityTrigger] IDurableActivityContext inputs)
{
    // parse input for student's major and year in university
    (string Major, int UniversityYear) studentInfo = inputs.GetInput<(string, int)>();

    // retrieve and return course recommendations by major and university year
    return new
    {
        major = studentInfo.Major,
        universityYear = studentInfo.UniversityYear,
        recommendedCourses = new []
        {
            "Introduction to .NET Programming",
            "Introduction to Linux",
            "Becoming an Entrepreneur"
        }
    };
}

```

Next steps

[Orchestrator code constraints](#)

Orchestrator function code constraints

10/5/2022 • 8 minutes to read • [Edit Online](#)

Durable Functions is an extension of [Azure Functions](#) that lets you build stateful apps. You can use an [orchestrator function](#) to orchestrate the execution of other durable functions within a function app. Orchestrator functions are stateful, reliable, and potentially long-running.

Orchestrator code constraints

Orchestrator functions use [event sourcing](#) to ensure reliable execution and to maintain local variable state. The [replay behavior](#) of orchestrator code creates constraints on the type of code that you can write in an orchestrator function. For example, orchestrator functions must be *deterministic*: an orchestrator function will be replayed multiple times, and it must produce the same result each time.

Using deterministic APIs

This section provides some simple guidelines that help ensure your code is deterministic.

Orchestrator functions can call any API in their target languages. However, it's important that orchestrator functions call only deterministic APIs. A *deterministic API* is an API that always returns the same value given the same input, no matter when or how often it's called.

The following sections provide guidance on APIs and patterns that you should avoid because they are *not* deterministic. These restrictions apply only to orchestrator functions. Other function types don't have such restrictions.

NOTE

Several types of code constraints are described below. This list is unfortunately not comprehensive and some use cases might not be covered. The most important thing to consider when writing orchestrator code is whether an API you're using is deterministic. Once you're comfortable with thinking this way, it's easy to understand which APIs are safe to use and which are not without needing to refer to this documented list.

Dates and times

APIs that return the current date or time are nondeterministic and should never be used in orchestrator functions. This is because each orchestrator function replay will produce a different value. You should instead use the Durable Functions equivalent API for getting the current date or time, which remains consistent across replays.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

Do not use `DateTime.Now`, `DateTime.UtcNow`, or equivalent APIs for getting the current time. Classes such as `Stopwatch` should also be avoided. For .NET in-process orchestrator functions, use the `IDurableOrchestrationContext.CurrentUtcDateTime` property to get the current time. For .NET isolated orchestrator functions, use the `TaskOrchestrationContext.CurrentDateTimeUtc` property to get the current time.

```
DateTime startTime = context.CurrentUtcDateTime;  
// do some work  
TimeSpan totalTime = context.CurrentUtcDateTime.Subtract(startTime);
```

GUIDs and UUIDs

APIs that return a random GUID or UUID are nondeterministic because the generated value is different for each replay. Depending on which language you use, a built-in API for generating deterministic GUIDs or UUIDs may be available. Otherwise, use an activity function to return a randomly generated GUID or UUID.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

Do not use APIs like `Guid.NewGuid()` to generate random GUIDs. Instead, use the context object's `NewGuid()` API to generate a random GUID that's safe for orchestrator replay.

```
Guid randomGuid = context.NewGuid();
```

NOTE

GUIDs generated with orchestration context APIs are [Type 5 UUIDs](#).

Random numbers

Use an activity function to return random numbers to an orchestrator function. The return values of activity functions are always safe for replay because they are saved into the orchestration history.

Alternatively, a random number generator with a fixed seed value can be used directly in an orchestrator function. This approach is safe as long as the same sequence of numbers is generated for each orchestration replay.

Bindings

An orchestrator function must not use any bindings, including even the [orchestration client](#) and [entity client](#) bindings. Always use input and output bindings from within a client or activity function. This is important because orchestrator functions may be replayed multiple times, causing nondeterministic and duplicate I/O with external systems.

Static variables

Avoid using static variables in orchestrator functions because their values can change over time, resulting in nondeterministic runtime behavior. Instead, use constants, or limit the use of static variables to activity functions.

NOTE

Even outside of orchestrator functions, using static variables in Azure Functions can be problematic for a variety of reasons since there's no guarantee that static state will persist across multiple function executions. Static variables should be avoided except in very specific usecases, such as best-effort in-memory caching in activity or entity functions.

Environment variables

Do not use environment variables in orchestrator functions. Their values can change over time, resulting in nondeterministic runtime behavior. If an orchestrator function needs configuration that's defined in an environment variable, you must pass the configuration value into the orchestrator function as an input or as the

return value of an activity function.

Network and HTTP

Use activity functions to make outbound network calls. If you need to make an HTTP call from your orchestrator function, you also can use the [durable HTTP APIs](#).

Thread-blocking APIs

Blocking APIs like "sleep" can cause performance and scale problems for orchestrator functions and should be avoided. In the Azure Functions Consumption plan, they can even result in unnecessary execution time charges. Use alternatives to blocking APIs when they're available. For example, use [Durable timers](#) to create delays that are safe for replay and don't count towards the execution time of an orchestrator function.

Async APIs

Orchestrator code must never start any async operation except those defined by the orchestration trigger's context object. For example, never use `Task.Run`, `Task.Delay`, and `HttpClient.SendAsync` in .NET or `setTimeout` and `setInterval` in JavaScript. An orchestrator function should only schedule async work using Durable SDK APIs, like scheduling activity functions. Any other type of async invocations should be done inside activity functions.

Async JavaScript functions

Always declare JavaScript orchestrator functions as synchronous generator functions. You must not declare JavaScript orchestrator functions as `async` because the Node.js runtime doesn't guarantee that asynchronous functions are deterministic.

Python coroutines

You must not declare Python orchestrator functions as coroutines. In other words, never declare Python orchestrator functions with the `async` keyword because coroutine semantics do not align with the Durable Functions replay model. You must always declare Python orchestrator functions as generators, meaning that you should expect the `context` API to use `yield` instead of `await`.

.NET threading APIs

The Durable Task Framework runs orchestrator code on a single thread and can't interact with any other threads. Running async continuations on a worker pool thread an orchestration's execution can result in nondeterministic execution or deadlocks. For this reason, orchestrator functions should almost never use threading APIs. For example, never use `ConfigureAwait(continueOnCapturedContext: false)` in an orchestrator function. This ensures that task continuations run on the orchestrator function's original `SynchronizationContext`.

NOTE

The Durable Task Framework attempts to detect accidental use of non-orchestrator threads in orchestrator functions. If it finds a violation, the framework throws a `NonDeterministicOrchestrationException` exception. However, this detection behavior won't catch all violations, and you shouldn't depend on it.

Versioning

A durable orchestration might run continuously for days, months, years, or even [eternally](#). Any code updates made to Durable Functions apps that affect unfinished orchestrations might break the orchestrations' replay behavior. That's why it's important to plan carefully when making updates to code. For a more detailed description of how to version your code, see the [versioning article](#).

Durable tasks

NOTE

This section describes internal implementation details of the Durable Task Framework. You can use durable functions without knowing this information. It is intended only to help you understand the replay behavior.

Tasks that can safely wait in orchestrator functions are occasionally referred to as *durable tasks*. The Durable Task Framework creates and manages these tasks. Examples are the tasks returned by `CallActivityAsync`, `WaitForExternalEvent`, and `CreateTimer` in .NET orchestrator functions.

These durable tasks are internally managed by a list of `TaskCompletionSource` objects in .NET. During replay, these tasks are created as part of orchestrator code execution. They're finished as the dispatcher enumerates the corresponding history events.

The tasks are executed synchronously using a single thread until all the history has been replayed. Durable tasks that aren't finished by the end of history replay have appropriate actions carried out. For example, a message might be enqueued to call an activity function.

This section's description of runtime behavior should help you understand why an orchestrator function can't use `await` or `yield` in a nondurable task. There are two reasons: the dispatcher thread can't wait for the task to finish, and any callback by that task might potentially corrupt the tracking state of the orchestrator function. Some runtime checks are in place to help detect these violations.

To learn more about how the Durable Task Framework executes orchestrator functions, consult the [Durable Task source code on GitHub](#). In particular, see `TaskOrchestrationExecutor.cs` and `TaskOrchestrationContext.cs`.

Next steps

[Learn how to invoke sub-orchestrations](#)

[Learn how to handle versioning](#)

Sub-orchestrations in Durable Functions (Azure Functions)

10/5/2022 • 3 minutes to read • [Edit Online](#)

In addition to calling activity functions, orchestrator functions can call other orchestrator functions. For example, you can build a larger orchestration out of a library of smaller orchestrator functions. Or you can run multiple instances of an orchestrator function in parallel.

An orchestrator function can call another orchestrator function using the "*call-sub-orchestrator*" API. The [Error Handling & Compensation](#) article has more information on automatic retry.

Sub-orchestrator functions behave just like activity functions from the caller's perspective. They can return a value, throw an exception, and can be awaited by the parent orchestrator function.

NOTE

Sub-orchestrations are not yet supported in PowerShell.

Example

The following example illustrates an IoT ("Internet of Things") scenario where there are multiple devices that need to be provisioned. The following function represents the provisioning workflow that needs to be executed for each device:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

```
public static async Task DeviceProvisioningOrchestration(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    string deviceId = context.GetInput<string>();

    // Step 1: Create an installation package in blob storage and return a SAS URL.
    Uri sasUrl = await context.CallActivityAsync<Uri>("CreateInstallationPackage", deviceId);

    // Step 2: Notify the device that the installation package is ready.
    await context.CallActivityAsync("SendPackageUrlToDevice", Tuple.Create(deviceId, sasUrl));

    // Step 3: Wait for the device to acknowledge that it has downloaded the new package.
    await context.WaitForExternalEvent<bool>("DownloadCompletedAck");

    // Step 4: ...
}
```

This orchestrator function can be used as-is for one-off device provisioning or it can be part of a larger orchestration. In the latter case, the parent orchestrator function can schedule instances of `DeviceProvisioningOrchestration` using the "*call-sub-orchestrator*" API.

Here is an example that shows how to run multiple orchestrator functions in parallel.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

```
[FunctionName("ProvisionNewDevices")]
public static async Task ProvisionNewDevices(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    string[] deviceIds = await context.CallActivityAsync<string[]>("GetNewDeviceIds");

    // Run multiple device provisioning flows in parallel
    var provisioningTasks = new List<Task>();
    foreach (string deviceId in deviceIds)
    {
        Task provisionTask = context.CallSubOrchestratorAsync("DeviceProvisioningOrchestration", deviceId);
        provisioningTasks.Add(provisionTask);
    }

    await Task.WhenAll(provisioningTasks);

    // ...
}
```

NOTE

The previous C# examples are for Durable Functions 2.x. For Durable Functions 1.x, you must use `DurableOrchestrationContext` instead of `IDurableOrchestrationContext`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

NOTE

Sub-orchestrations must be defined in the same function app as the parent orchestration. If you need to call and wait for orchestrations in another function app, consider using the built-in support for HTTP APIs and the HTTP 202 polling consumer pattern. For more information, see the [HTTP Features](#) topic.

Next steps

Learn how to set a custom orchestration status

Custom orchestration status in Durable Functions (Azure Functions)

10/5/2022 • 8 minutes to read • [Edit Online](#)

Custom orchestration status lets you set a custom status value for your orchestrator function. This status is provided via the [HTTP GetStatus API](#) or the equivalent [SDK API](#) on the orchestration client object.

Sample use cases

Visualize progress

Clients can poll the status end point and display a progress UI that visualizes the current execution stage. The following sample demonstrates progress sharing:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

NOTE

These C# examples are written for Durable Functions 2.x and are not compatible with Durable Functions 1.x. For more information about the differences between versions, see the [Durable Functions versions](#) article.

```
[FunctionName("E1_HelloSequence")]
public static async Task<List<string>> Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    var outputs = new List<string>();

    outputs.Add(await context.CallActivityAsync<string>("E1_SayHello", "Tokyo"));
    context.SetCustomStatus("Tokyo");
    outputs.Add(await context.CallActivityAsync<string>("E1_SayHello", "Seattle"));
    context.SetCustomStatus("Seattle");
    outputs.Add(await context.CallActivityAsync<string>("E1_SayHello", "London"));
    context.SetCustomStatus("London");

    // returns ["Hello Tokyo!", "Hello Seattle!", "Hello London!"]
    return outputs;
}

[FunctionName("E1_SayHello")]
public static string SayHello([ActivityTrigger] string name)
{
    return $"Hello {name}!";
}
```

And then the client will receive the output of the orchestration only when `CustomStatus` field is set to "London":

- [C#](#)
- [JavaScript](#)

- [Python](#)
- [PowerShell](#)
- [Java](#)

```
[FunctionName("HttpStart")]
public static async Task<HttpResponseMessage> Run(
    [HttpTrigger(AuthorizationLevel.Function, methods: "post", Route = "orchestrators/{functionName}")]
    HttpRequestMessage req,
    [DurableClient] IDurableOrchestrationClient starter,
    string functionName,
    ILogger log)
{
    // Function input comes from the request content.
    dynamic eventData = await req.Content.ReadAsAsync<object>();
    string instanceId = await starter.StartNewAsync(functionName, (string)eventData);

    log.LogInformation($"Started orchestration with ID = '{instanceId}'.");

    DurableOrchestrationStatus durableOrchestrationStatus = await starter.GetStatusAsync(instanceId);
    while (durableOrchestrationStatus.CustomStatus.ToString() != "London")
    {
        await Task.Delay(200);
        durableOrchestrationStatus = await starter.GetStatusAsync(instanceId);
    }

    HttpResponseMessage httpResponseMessage = new HttpResponseMessage(HttpStatusCode.OK)
    {
        Content = new StringContent(JsonConvert.SerializeObject(durableOrchestrationStatus))
    };

    return httpResponseMessage;
}
}
```

Output customization

Another interesting scenario is segmenting users by returning customized output based on unique characteristics or interactions. With the help of custom orchestration status, the client-side code will stay generic. All main modifications will happen on the server side as shown in the following sample:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```

[FunctionName("CityRecommender")]
public static void Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    int userChoice = context.GetInput<int>();

    switch (userChoice)
    {
        case 1:
            context.SetCustomStatus(new
            {
                recommendedCities = new[] {"Tokyo", "Seattle"},
                recommendedSeasons = new[] {"Spring", "Summer"}
            });
            break;
        case 2:
            context.SetCustomStatus(new
            {
                recommendedCities = new[] {"Seattle, London"},
                recommendedSeasons = new[] {"Summer"}
            });
            break;
        case 3:
            context.SetCustomStatus(new
            {
                recommendedCities = new[] {"Tokyo, London"},
                recommendedSeasons = new[] {"Spring", "Summer"}
            });
            break;
    }

    // Wait for user selection and refine the recommendation
}

```

Instruction specification

The orchestrator can provide unique instructions to the clients via the custom state. The custom status instructions will be mapped to the steps in the orchestration code:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```
[FunctionName("ReserveTicket")]
public static async Task<bool> Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    string userId = context.GetInput<string>();

    int discount = await context.CallActivityAsync<int>("CalculateDiscount", userId);

    context.SetCustomStatus(new
    {
        discount = discount,
        discountTimeout = 60,
        bookingUrl = "https://www.myawesomebookingweb.com",
    });

    bool isBookingConfirmed = await context.WaitForExternalEvent<bool>("BookingConfirmed");

    context.SetCustomStatus(isBookingConfirmed
        ? new {message = "Thank you for confirming your booking."}
        : new {message = "The booking was not confirmed on time. Please try again."});

    return isBookingConfirmed;
}
```

Querying custom status with HTTP

The following example shows how custom status values can be queried using the built-in HTTP APIs.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```
public static async Task SetStatusTest([OrchestrationTrigger] IDurableOrchestrationContext context)
{
    // ...do work...

    // update the status of the orchestration with some arbitrary data
    var customStatus = new { nextActions = new [] {"A", "B", "C"}, foo = 2, };
    context.SetCustomStatus(customStatus);

    // ...do more work...
}
```

While the orchestration is running, external clients can fetch this custom status:

```
GET /runtime/webhooks/durabletask/instances/instance123
```

Clients will get the following response:

```
{  
  "runtimeStatus": "Running",  
  "input": null,  
  "customStatus": { "nextActions": ["A", "B", "C"], "foo": 2 },  
  "output": null,  
  "createdTime": "2019-10-06T18:30:24Z",  
  "lastUpdatedTime": "2019-10-06T19:40:30Z"  
}
```

WARNING

The custom status payload is limited to 16 KB of UTF-16 JSON text. We recommend you use external storage if you need a larger payload.

Next steps

[Learn about durable timers](#)

Timers in Durable Functions (Azure Functions)

10/5/2022 • 5 minutes to read • [Edit Online](#)

[Durable Functions](#) provides *durable timers* for use in orchestrator functions to implement delays or to set up timeouts on async actions. Durable timers should be used in orchestrator functions instead of "sleep" or "delay" APIs that may be built into the language.

Durable timers are tasks that are created using the appropriate "create timer" API for the provided language, as shown below, and take either a due time or a duration as an argument.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```
// Put the orchestrator to sleep for 72 hours
DateTime dueTime = context.CurrentUtcDateTime.AddHours(72);
await context.CreateTimer(dueTime, CancellationToken.None);
```

When you "await" the timer task, the orchestrator function will sleep until the specified expiration time.

NOTE

Orchestrations will continue to process other incoming events while waiting for a timer task to expire.

Timer limitations

When you create a timer that expires at 4:30 pm UTC, the underlying Durable Task Framework enqueues a message that becomes visible only at 4:30 pm UTC. If the function app is scaled down to zero instances in the meantime, the newly visible timer message will ensure that the function app gets activated again on an appropriate VM.

NOTE

- Starting with [version 2.3.0](#) of the Durable Extension, Durable timers are unlimited for .NET apps. For JavaScript, Python, and PowerShell apps, as well as .NET apps using earlier versions of the extension, Durable timers are limited to six days. When you are using an older extension version or a non-.NET language runtime and need a delay longer than six days, use the timer APIs in a `while` loop to simulate a longer delay.
- Don't use built-in date/time APIs for getting the current time. When calculating a future date for a timer to expire, always use the orchestrator function's current time API. For more information, see the [orchestrator function code constraints](#) article.

Usage for delay

The following example illustrates how to use durable timers for delaying execution. The example is issuing a billing notification every day for 10 days.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```
[FunctionName("BillingIssuer")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    for (int i = 0; i < 10; i++)
    {
        DateTime deadline = context.CurrentUtcDateTime.Add(TimeSpan.FromDays(1));
        await context.CreateTimer(deadline, CancellationToken.None);
        await context.CallActivityAsync("SendBillingEvent");
    }
}
```

NOTE

The previous C# example targets Durable Functions 2.x. For Durable Functions 1.x, you must use `DurableOrchestrationContext` instead of `IDurableOrchestrationContext`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

WARNING

Avoid infinite loops in orchestrator functions. For information about how to safely and efficiently implement infinite loop scenarios, see [Eternal Orchestrations](#).

Usage for timeout

This example illustrates how to use durable timers to implement timeouts.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```
[FunctionName("TryGetQuote")]
public static async Task<bool> Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    TimeSpan timeout = TimeSpan.FromSeconds(30);
    DateTime deadline = context.CurrentUtcDateTime.Add(timeout);

    using (var cts = new CancellationTokenSource())
    {
        Task activityTask = context.CallActivityAsync("GetQuote");
        Task timeoutTask = context.CreateTimer(deadline, cts.Token);

        Task winner = await Task.WhenAny(activityTask, timeoutTask);
        if (winner == activityTask)
        {
            // success case
            cts.Cancel();
            return true;
        }
        else
        {
            // timeout case
            return false;
        }
    }
}
```

NOTE

The previous C# example targets Durable Functions 2.x. For Durable Functions 1.x, you must use `DurableOrchestrationContext` instead of `IDurableOrchestrationContext`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

WARNING

In .NET, JavaScript, Python, and PowerShell, you must cancel any created durable timers if your code will not wait for them to complete. See the examples above for how to cancel pending timers. The Durable Task Framework will not change an orchestration's status to "Completed" until all outstanding tasks, including durable timer tasks, are either completed or canceled.

This cancellation mechanism using the *when-any* pattern doesn't terminate in-progress activity function or sub-orchestration executions. Rather, it simply allows the orchestrator function to ignore the result and move on. If your function app uses the Consumption plan, you'll still be billed for any time and memory consumed by the abandoned activity function. By default, functions running in the Consumption plan have a timeout of five minutes. If this limit is exceeded, the Azure Functions host is recycled to stop all execution and prevent a runaway billing situation. The [function timeout is configurable](#).

For a more in-depth example of how to implement timeouts in orchestrator functions, see the [Human Interaction & Timeouts - Phone Verification](#) article.

Next steps

[Learn how to raise and handle external events](#)

Handling external events in Durable Functions (Azure Functions)

10/5/2022 • 7 minutes to read • [Edit Online](#)

Orchestrator functions have the ability to wait and listen for external events. This feature of [Durable Functions](#) is often useful for handling human interaction or other external triggers.

NOTE

External events are one-way asynchronous operations. They are not suitable for situations where the client sending the event needs a synchronous response from the orchestrator function.

Wait for events

The “*wait-for-external-event*” API of the [orchestration trigger binding](#) allows an orchestrator function to asynchronously wait and listen for an event delivered by an external client. The listening orchestrator function declares the *name* of the event and the *shape of the data* it expects to receive.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```
[FunctionName("BudgetApproval")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    bool approved = await context.WaitForExternalEvent<bool>("Approval");
    if (approved)
    {
        // approval granted - do the approved action
    }
    else
    {
        // approval denied - send a notification
    }
}
```

NOTE

The previous C# code is for Durable Functions 2.x. For Durable Functions 1.x, you must use

`DurableOrchestrationContext` instead of `IDurableOrchestrationContext`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

The preceding example listens for a specific single event and takes action when it's received.

You can listen for multiple events concurrently, like in the following example, which waits for one of three possible event notifications.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```
[FunctionName("Select")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    var event1 = context.WaitForExternalEvent<float>("Event1");
    var event2 = context.WaitForExternalEvent<bool>("Event2");
    var event3 = context.WaitForExternalEvent<int>("Event3");

    var winner = await Task.WhenAny(event1, event2, event3);
    if (winner == event1)
    {
        // ...
    }
    else if (winner == event2)
    {
        // ...
    }
    else if (winner == event3)
    {
        // ...
    }
}
```

NOTE

The previous C# code is for Durable Functions 2.x. For Durable Functions 1.x, you must use

`DurableOrchestrationContext` instead of `IDurableOrchestrationContext`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

The previous example listens for *any* of multiple events. It's also possible to wait for *all* events.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```
[FunctionName("NewBuildingPermit")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    string applicationId = context.GetInput<string>();

    var gate1 = context.WaitForExternalEvent("CityPlanningApproval");
    var gate2 = context.WaitForExternalEvent("FireDeptApproval");
    var gate3 = context.WaitForExternalEvent("BuildingDeptApproval");

    // all three departments must grant approval before a permit can be issued
    await Task.WhenAll(gate1, gate2, gate3);

    await context.CallActivityAsync("IssueBuildingPermit", applicationId);
}
```

NOTE

The previous code is for Durable Functions 2.x. For Durable Functions 1.x, you must use `DurableOrchestrationContext` instead of `IDurableOrchestrationContext`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

In .NET, if the event payload cannot be converted into the expected type `T`, an exception is thrown.

The “*wait-for-external-event*” API waits indefinitely for some input. The function app can be safely unloaded while waiting. If and when an event arrives for this orchestration instance, it is awakened automatically and immediately processes the event.

NOTE

If your function app uses the Consumption Plan, no billing charges are incurred while an orchestrator function is awaiting an external event task, no matter how long it waits.

Send events

You can use the “*raise-event*” API defined by the [orchestration client](#) binding to send an external event to an orchestration. You can also use the built-in [raise event HTTP API](#) to send an external event to an orchestration.

A raised event includes an *instance ID*, an *eventName*, and *eventData* as parameters. Orchestrator functions handle these events using the “*wait-for-external-event*” APIs. The *eventName* must match on both the sending and receiving ends in order for the event to be processed. The event data must also be JSON-serializable.

Internally, the “*raise-event*” mechanisms enqueue a message that gets picked up by the waiting orchestrator function. If the instance is not waiting on the specified *event name*, the event message is added to an in-memory queue. If the orchestration instance later begins listening for that *event name*, it will check the queue for event messages.

NOTE

If there is no orchestration instance with the specified *instance ID*, the event message is discarded.

Below is an example queue-triggered function that sends an “Approval” event to an orchestrator function instance. The orchestration instance ID comes from the body of the queue message.

- [C#](#)

- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```
[FunctionName("ApprovalQueueProcessor")]
public static async Task Run(
    [QueueTrigger("approval-queue")] string instanceId,
    [DurableClient] IDurableOrchestrationClient client)
{
    await client.RaiseEventAsync(instanceId, "Approval", true);
}
```

NOTE

The previous C# code is for Durable Functions 2.x. For Durable Functions 1.x, you must use `OrchestrationClient` attribute instead of the `DurableClient` attribute, and you must use the `DurableOrchestrationClient` parameter type instead of `IDurableOrchestrationClient`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

Internally, the "*raise-event*" API enqueues a message that gets picked up by the waiting orchestrator function. If the instance is not waiting on the specified *event name*, the event message is added to an in-memory buffer. If the orchestration instance later begins listening for that *event name*, it will check the buffer for event messages and trigger the task that was waiting for it.

NOTE

If there is no orchestration instance with the specified *instance ID*, the event message is discarded.

HTTP

The following is an example of an HTTP request that raises an "Approval" event to an orchestration instance.

```
POST /runtime/webhooks/durabletask/instances/MyInstanceId/raiseEvent/Approval&code=XXX
Content-Type: application/json

"true"
```

In this case, the instance ID is hardcoded as *MyInstanceId*.

Next steps

[Learn how to implement error handling](#)

[Run a sample that waits for human interaction](#)

Handling errors in Durable Functions (Azure Functions)

10/5/2022 • 7 minutes to read • [Edit Online](#)

Durable Function orchestrations are implemented in code and can use the programming language's built-in error-handling features. There really aren't any new concepts you need to learn to add error handling and compensation into your orchestrations. However, there are a few behaviors that you should be aware of.

Errors in activity functions

Any exception that is thrown in an activity function is marshaled back to the orchestrator function and thrown as a `FunctionFailedException`. You can write error handling and compensation code that suits your needs in the orchestrator function.

For example, consider the following orchestrator function that transfers funds from one account to another:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```
[FunctionName("TransferFunds")]
public static async Task Run([OrchestrationTrigger] IDurableOrchestrationContext context)
{
    var transferDetails = context.GetInput<TransferOperation>();

    await context.CallActivityAsync("DebitAccount",
        new
    {
        Account = transferDetails.SourceAccount,
        Amount = transferDetails.Amount
    });

    try
    {
        await context.CallActivityAsync("CreditAccount",
            new
        {
            Account = transferDetails.DestinationAccount,
            Amount = transferDetails.Amount
        });
    }
    catch (Exception)
    {
        // Refund the source account.
        // Another try/catch could be used here based on the needs of the application.
        await context.CallActivityAsync("CreditAccount",
            new
        {
            Account = transferDetails.SourceAccount,
            Amount = transferDetails.Amount
        });
    }
}
```

NOTE

The previous C# examples are for Durable Functions 2.x. For Durable Functions 1.x, you must use `DurableOrchestrationContext` instead of `IDurableOrchestrationContext`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

If the first `CreditAccount` function call fails, the orchestrator function compensates by crediting the funds back to the source account.

Automatic retry on failure

When you call activity functions or sub-orchestration functions, you can specify an automatic retry policy. The following example attempts to call a function up to three times and waits 5 seconds between each retry:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```
[FunctionName("TimerOrchestratorWithRetry")]
public static async Task Run([OrchestrationTrigger] IDurableOrchestrationContext context)
{
    var retryOptions = new RetryOptions(
        firstRetryInterval: TimeSpan.FromSeconds(5),
        maxNumberOfAttempts: 3);

    await context.CallActivityWithRetryAsync("FlakyFunction", retryOptions, null);

    // ...
}
```

NOTE

The previous C# examples are for Durable Functions 2.x. For Durable Functions 1.x, you must use `DurableOrchestrationContext` instead of `IDurableOrchestrationContext`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

The activity function call in the previous example takes a parameter for configuring an automatic retry policy. There are several options for customizing the automatic retry policy:

- **Max number of attempts:** The maximum number of attempts. If set to 1, there will be no retry.
- **First retry interval:** The amount of time to wait before the first retry attempt.
- **Backoff coefficient:** The coefficient used to determine rate of increase of backoff. Defaults to 1.
- **Max retry interval:** The maximum amount of time to wait in between retry attempts.
- **Retry timeout:** The maximum amount of time to spend doing retries. The default behavior is to retry indefinitely.

Custom retry handlers

When using the .NET isolated worker or Java, you also have the option to implement retry handlers in code. This is useful when declarative retry policies are not expressive enough. For languages that don't support custom retry handlers, you still have the option of implementing retry policies using loops, exception handling, and timers for injecting delays between retries.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```

TaskOptions retryOptions = TaskOptions.FromRetryHandler(retryContext =>
{
    // Don't retry anything that derives from ApplicationException
    if (!retryContext.LastFailure.IsCausedBy<ApplicationException>())
    {
        return false;
    }

    // Quit after N attempts
    return retryContext.LastAttemptNumber < 3;
});

try
{
    await ctx.CallActivityAsync("FlakeyActivity", options: retryOptions);
}
catch (TaskFailedException)
{
    // Case when the retry handler returns false...
}

```

Function timeouts

You might want to abandon a function call within an orchestrator function if it's taking too long to complete. The proper way to do this today is by creating a [durable timer](#) with an "any" task selector, as in the following example:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```

[FunctionName("TimerOrchestrator")]
public static async Task<bool> Run([OrchestrationTrigger] IDurableOrchestrationContext context)
{
    TimeSpan timeout = TimeSpan.FromSeconds(30);
    DateTime deadline = context.CurrentUtcDateTime.Add(timeout);

    using (var cts = new CancellationTokenSource())
    {
        Task activityTask = context.CallActivityAsync("FlakyFunction");
        Task timeoutTask = context.CreateTimer(deadline, cts.Token);

        Task winner = await Task.WhenAny(activityTask, timeoutTask);
        if (winner == activityTask)
        {
            // success case
            cts.Cancel();
            return true;
        }
        else
        {
            // timeout case
            return false;
        }
    }
}

```

NOTE

The previous C# examples are for Durable Functions 2.x. For Durable Functions 1.x, you must use `DurableOrchestrationContext` instead of `IDurableOrchestrationContext`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

NOTE

This mechanism does not actually terminate in-progress activity function execution. Rather, it simply allows the orchestrator function to ignore the result and move on. For more information, see the [Timers](#) documentation.

Unhandled exceptions

If an orchestrator function fails with an unhandled exception, the details of the exception are logged and the instance completes with a `Failed` status.

Next steps

[Learn about eternal orchestrations](#)

[Learn how to diagnose problems](#)

Eternal orchestrations in Durable Functions (Azure Functions)

10/5/2022 • 3 minutes to read • [Edit Online](#)

Eternal orchestrations are orchestrator functions that never end. They are useful when you want to use [Durable Functions](#) for aggregators and any scenario that requires an infinite loop.

Orchestration history

As explained in the [orchestration history](#) topic, the Durable Task Framework keeps track of the history of each function orchestration. This history grows continuously as long as the orchestrator function continues to schedule new work. If the orchestrator function goes into an infinite loop and continuously schedules work, this history could grow critically large and cause significant performance problems. The *eternal orchestration* concept was designed to mitigate these kinds of problems for applications that need infinite loops.

Resetting and restarting

Instead of using infinite loops, orchestrator functions reset their state by calling the *continue-as-new* method of the [orchestration trigger binding](#). This method takes a JSON-serializable parameter, which becomes the new input for the next orchestrator function generation.

When *continue-as-new* is called, the orchestration instance restarts itself with the new input value. The same instance ID is kept, but the orchestrator function's history is reset.

NOTE

The Durable Task Framework maintains the same instance ID but internally creates a new *execution ID* for the orchestrator function that gets reset by *continue-as-new*. This execution ID is not exposed externally, but it may be useful to know about when debugging orchestration execution.

Periodic work example

One use case for eternal orchestrations is code that needs to do periodic work indefinitely.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```
[FunctionName("Periodic_Cleanup_Loop")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    await context.CallActivityAsync("DoCleanup", null);

    // sleep for one hour between cleanups
    DateTime nextCleanup = context.CurrentUtcDateTime.AddHours(1);
    await context.CreateTimer(nextCleanup, CancellationToken.None);

    context.ContinueAsNew(null);
}
```

NOTE

The previous C# example is for Durable Functions 2.x. For Durable Functions 1.x, you must use `DurableOrchestrationContext` instead of `IDurableOrchestrationContext`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

The difference between this example and a timer-triggered function is that cleanup trigger times here are not based on a schedule. For example, a CRON schedule that executes a function every hour will execute it at 1:00, 2:00, 3:00 etc. and could potentially run into overlap issues. In this example, however, if the cleanup takes 30 minutes, then it will be scheduled at 1:00, 2:30, 4:00, etc. and there is no chance of overlap.

Starting an eternal orchestration

Use the *start-new* or *schedule-new* durable client method to start an eternal orchestration, just like you would any other orchestration function.

NOTE

If you need to ensure a singleton eternal orchestration is running, it's important to maintain the same instance `id` when starting the orchestration. For more information, see [Instance Management](#).

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```
[FunctionName("Trigger_Eternal_Orchestration")]
public static async Task<HttpResponseMessage> OrchestrationTrigger(
    [HttpTrigger(AuthorizationLevel.Function, "post", Route = null)] HttpRequestMessage request,
    [DurableClient] IDurableOrchestrationClient client)
{
    string instanceId = "StaticId";

    await client.StartNewAsync("Periodic_Cleanup_Loop", instanceId);
    return client.CreateCheckStatusResponse(request, instanceId);
}
```

NOTE

The previous code is for Durable Functions 2.x. For Durable Functions 1.x, you must use `OrchestrationClient` attribute instead of the `DurableClient` attribute, and you must use the `DurableOrchestrationClient` parameter type instead of `IDurableOrchestrationClient`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

Exit from an eternal orchestration

If an orchestrator function needs to eventually complete, then all you need to do is *not* call `ContinueAsNew` and let the function exit.

If an orchestrator function is in an infinite loop and needs to be stopped, use the *terminate* API of the [orchestration client binding](#) to stop it. For more information, see [Instance Management](#).

Next steps

[Learn how to implement singleton orchestrations](#)

Singleton orchestrators in Durable Functions (Azure Functions)

10/5/2022 • 3 minutes to read • [Edit Online](#)

For background jobs, you often need to ensure that only one instance of a particular orchestrator runs at a time. You can ensure this kind of singleton behavior in [Durable Functions](#) by assigning a specific instance ID to an orchestrator when creating it.

Singleton example

The following example shows an HTTP-trigger function that creates a singleton background job orchestration. The code ensures that only one instance exists for a specified instance ID.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

```
[FunctionName("HttpStartSingle")]
public static async Task<HttpResponseMessage> RunSingle(
    [HttpTrigger(AuthorizationLevel.Function, methods: "post", Route =
"orchestrators/{functionName}/{instanceId}")] HttpRequestMessage req,
    [DurableClient] IDurableOrchestrationClient starter,
    string functionName,
    string instanceId,
    ILogger log)
{
    // Check if an instance with the specified ID already exists or an existing one stopped
    running(completed/failed/terminated).
    var existingInstance = await starter.GetStatusAsync(instanceId);
    if (existingInstance == null
        || existingInstance.RuntimeStatus == OrchestrationRuntimeStatus.Completed
        || existingInstance.RuntimeStatus == OrchestrationRuntimeStatus.Failed
        || existingInstance.RuntimeStatus == OrchestrationRuntimeStatus.Terminated)
    {
        // An instance with the specified ID doesn't exist or an existing one stopped running, create one.
        dynamic eventData = await req.Content.ReadAsAsync<object>();
        await starter.StartNewAsync(functionName, instanceId, eventData);
        log.LogInformation($"Started orchestration with ID = '{instanceId}'.");
        return starter.CreateCheckStatusResponse(req, instanceId);
    }
    else
    {
        // An instance with the specified ID exists or an existing one still running, don't create one.
        return new HttpResponseMessage(HttpStatusCode.Conflict)
        {
            Content = new StringContent($"An instance with ID '{instanceId}' already exists."),
        };
    }
}
```

NOTE

The previous C# code is for Durable Functions 2.x. For Durable Functions 1.x, you must use `OrchestrationClient` attribute instead of the `DurableClient` attribute, and you must use the `DurableOrchestrationClient` parameter type instead of `IDurableOrchestrationClient`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

By default, instance IDs are randomly generated GUIDs. In the previous example, however, the instance ID is passed in route data from the URL. The code then fetches the orchestration instance metadata to check if an instance having the specified ID is already running. If no such instance is running, a new instance is created with that ID.

NOTE

There is a potential race condition in this sample. If two instances of `HttpStartSingle` execute concurrently, both function calls will report success, but only one orchestration instance will actually start. Depending on your requirements, this may have undesirable side effects.

The implementation details of the orchestrator function don't actually matter. It could be a regular orchestrator function that starts and completes, or it could be one that runs forever (that is, an [Eternal Orchestration](#)). The important point is that there is only ever one instance running at a time.

Next steps

[Learn about the native HTTP features of orchestrations](#)

HTTP Features

10/5/2022 • 11 minutes to read • [Edit Online](#)

Durable Functions has several features that make it easy to incorporate durable orchestrations and entities into HTTP workflows. This article goes into detail about some of those features.

Exposing HTTP APIs

Orchestrations and entities can be invoked and managed using HTTP requests. The Durable Functions extension exposes built-in HTTP APIs. It also provides APIs for interacting with orchestrations and entities from within HTTP-triggered functions.

Built-in HTTP APIs

The Durable Functions extension automatically adds a set of HTTP APIs to the Azure Functions host. With these APIs, you can interact with and manage orchestrations and entities without writing any code.

The following built-in HTTP APIs are supported.

- [Start new orchestration](#)
- [Query orchestration instance](#)
- [Terminate orchestration instance](#)
- [Send an external event to an orchestration](#)
- [Purge orchestration history](#)
- [Send an operation event to an entity](#)
- [Get the state of an entity](#)
- [Query the list of entities](#)

See the [HTTP APIs article](#) for a full description of all the built-in HTTP APIs exposed by the Durable Functions extension.

HTTP API URL discovery

The [orchestration client binding](#) exposes APIs that can generate convenient HTTP response payloads. For example, it can create a response containing links to management APIs for a specific orchestration instance. The following examples show an HTTP-trigger function that demonstrates how to use this API for a new orchestration instance:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```

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

using System.Net.Http;
using System.Threading.Tasks;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.DurableTask;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Extensions.Logging;

namespace VSSample
{
    public static class HttpStart
    {
        [FunctionName("HttpStart")]
        public static async Task<HttpResponseMessage> Run(
            [HttpTrigger(AuthorizationLevel.Function, methods: "post", Route =
"orchestrators/{functionName}")] HttpRequestMessage req,
            [DurableClient] IDurableClient starter,
            string functionName,
            ILogger log)
        {
            // Function input comes from the request content.
            object eventData = await req.Content.ReadAsAsync<object>();
            string instanceId = await starter.StartNewAsync(functionName, eventData);

            log.LogInformation($"Started orchestration with ID = '{instanceId}'.");

            return starter.CreateCheckStatusResponse(req, instanceId);
        }
    }
}

```

Starting an orchestrator function by using the HTTP-trigger functions shown previously can be done using any HTTP client. The following cURL command starts an orchestrator function named `DoWork`:

```
curl -X POST https://localhost:7071/orchestrators/DoWork -H "Content-Length: 0" -i
```

Next is an example response for an orchestration that has `abc123` as its ID. Some details have been removed for clarity.

```

HTTP/1.1 202 Accepted
Content-Type: application/json; charset=utf-8
Location: http://localhost:7071/runtime/webhooks/durabletask/instances/abc123?code=XXX
Retry-After: 10

{
    "id": "abc123",
    "purgeHistoryDeleteUri": "http://localhost:7071/runtime/webhooks/durabletask/instances/abc123?code=XXX",
    "sendEventPostUri":
    "http://localhost:7071/runtime/webhooks/durabletask/instances/abc123/raiseEvent/{eventName}?code=XXX",
    "statusQueryGetUri": "http://localhost:7071/runtime/webhooks/durabletask/instances/abc123?code=XXX",
    "terminatePostUri": "http://localhost:7071/runtime/webhooks/durabletask/instances/abc123/terminate?
reason={text}&code=XXX"
}

```

In the previous example, each of the fields ending in `Uri` corresponds to a built-in HTTP API. You can use these APIs to manage the target orchestration instance.

NOTE

The format of the webhook URLs depends on which version of the Azure Functions host you are running. The previous example is for the Azure Functions 2.0 host.

For a description of all built-in HTTP APIs, see the [HTTP API reference](#).

Async operation tracking

The HTTP response mentioned previously is designed to help implement long-running HTTP async APIs with Durable Functions. This pattern is sometimes referred to as the *polling consumer pattern*. The client/server flow works as follows:

1. The client issues an HTTP request to start a long-running process like an orchestrator function.
2. The target HTTP trigger returns an HTTP 202 response with a Location header that has the value "statusQueryGetUri".
3. The client polls the URL in the Location header. The client continues to see HTTP 202 responses with a Location header.
4. When the instance finishes or fails, the endpoint in the Location header returns HTTP 200.

This protocol allows coordination of long-running processes with external clients or services that can poll an HTTP endpoint and follow the Location header. Both the client and server implementations of this pattern are built into the Durable Functions HTTP APIs.

NOTE

By default, all HTTP-based actions provided by [Azure Logic Apps](#) support the standard asynchronous operation pattern. This capability makes it possible to embed a long-running durable function as part of a Logic Apps workflow. You can find more details on Logic Apps support for asynchronous HTTP patterns in the [Azure Logic Apps workflow actions and triggers documentation](#).

NOTE

Interactions with orchestrations can be done from any function type, not just HTTP-triggered functions.

For more information on how to manage orchestrations and entities using client APIs, see the [Instance management article](#).

Consuming HTTP APIs

As described in the [orchestrator function code constraints](#), orchestrator functions can't do I/O directly. Instead, they typically call [activity functions](#) that do I/O operations.

Starting with Durable Functions 2.0, orchestrations can natively consume HTTP APIs by using the [orchestration trigger binding](#).

The following example code shows an orchestrator function making an outbound HTTP request:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```
[FunctionName("CheckSiteAvailable")]
public static async Task CheckSiteAvailable(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    Uri url = context.GetInput<Uri>();

    // Makes an HTTP GET request to the specified endpoint
    DurableHttpResponse response =
        await context.CallHttpAsync(HttpMethod.Get, url);

    if (response.StatusCode >= 400)
    {
        // handling of error codes goes here
    }
}
```

By using the "call HTTP" action, you can do the following actions in your orchestrator functions:

- Call HTTP APIs directly from orchestration functions, with some limitations that are mentioned later.
- Automatically support client-side HTTP 202 status polling patterns.
- Use [Azure Managed Identities](#) to make authorized HTTP calls to other Azure endpoints.

The ability to consume HTTP APIs directly from orchestrator functions is intended as a convenience for a certain set of common scenarios. You can implement all of these features yourself using activity functions. In many cases, activity functions might give you more flexibility.

HTTP 202 handling

The "call HTTP" API can automatically implement the client side of the polling consumer pattern. If a called API returns an HTTP 202 response with a Location header, the orchestrator function automatically polls the Location resource until receiving a response other than 202. This response will be the response returned to the orchestrator function code.

NOTE

1. Orchestrator functions also natively support the server-side polling consumer pattern, as described in [Async operation tracking](#). This support means that orchestrations in one function app can easily coordinate the orchestrator functions in other function apps. This is similar to the [sub-orchestration](#) concept, but with support for cross-app communication. This support is particularly useful for microservice-style app development.
2. Due to a temporary limitation, the built-in HTTP polling pattern is not currently available in JavaScript/TypeScript and Python.

Managed identities

Durable Functions natively supports calls to APIs that accept Azure Active Directory (Azure AD) tokens for authorization. This support uses [Azure managed identities](#) to acquire these tokens.

The following code is an example of an orchestrator function. The function makes authenticated calls to restart a virtual machine by using the Azure Resource Manager [virtual machines REST API](#).

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```

[FunctionName("RestartVm")]
public static async Task RunOrchestrator(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    string subscriptionId = "mySubId";
    string resourceGroup = "myRG";
    string vmName = "myVM";
    string apiVersion = "2019-03-01";

    // Automatically fetches an Azure AD token for resource = https://management.core.windows.net/.default
    // and attaches it to the outgoing Azure Resource Manager API call.
    var restartRequest = new DurableHttpRequest(
        HttpMethod.Post,
        new Uri($"https://management.azure.com/subscriptions/{subscriptionId}/resourceGroups/{resourceGroup}/providers/Microsoft.Compute/virtualMachines/{vmName}/restart?api-version={apiVersion}"),
        tokenSource: new ManagedIdentityTokenSource("https://management.core.windows.net/.default"));
    DurableHttpResponse restartResponse = await context.CallHttpAsync(restartRequest);
    if (restartResponse.StatusCode != HttpStatusCode.OK)
    {
        throw new ArgumentException($"Failed to restart VM: {restartResponse.StatusCode}:{restartResponse.Content}");
    }
}

```

In the previous example, the `tokenSource` parameter is configured to acquire Azure AD tokens for [Azure Resource Manager](#). The tokens are identified by the resource URI `https://management.core.windows.net/.default`. The example assumes that the current function app either is running locally or was deployed as a function app with a managed identity. The local identity or the managed identity is assumed to have permission to manage VMs in the specified resource group `myRG`.

At runtime, the configured token source automatically returns an OAuth 2.0 access token. The source then adds the token as a bearer token to the Authorization header of the outgoing request. This model is an improvement over manually adding authorization headers to HTTP requests for the following reasons:

- Token refresh is handled automatically. You don't need to worry about expired tokens.
- Tokens are never stored in the durable orchestration state.
- You don't need to write any code to manage token acquisition.

You can find a more complete example in the [precompiled C# RestartVMs sample](#).

Managed identities aren't limited to Azure resource management. You can use managed identities to access any API that accepts Azure AD bearer tokens, including Azure services from Microsoft and web apps from partners. A partner's web app can even be another function app. For a list of Azure services from Microsoft that support authentication with Azure AD, see [Azure services that support Azure AD authentication](#).

Limitations

The built-in support for calling HTTP APIs is a convenience feature. It's not appropriate for all scenarios.

HTTP requests sent by orchestrator functions and their responses are [serialized and persisted](#) as messages in the Durable Functions storage provider. This persistent queuing behavior ensures HTTP calls are [reliable and safe for orchestration replay](#). However, the persistent queuing behavior also has limitations:

- Each HTTP request involves additional latency when compared to a native HTTP client.
- Depending on the [configured storage provider](#), large request or response messages can significantly degrade orchestration performance. For example, when using Azure Storage, HTTP payloads that are too large to fit into Azure Queue messages are compressed and stored in Azure Blob storage.
- Streaming, chunked, and binary payloads aren't supported.
- The ability to customize the behavior of the HTTP client is limited.

If any of these limitations might affect your use case, consider instead using activity functions and language-specific HTTP client libraries to make outbound HTTP calls.

NOTE

If you are a .NET developer, you might wonder why this feature uses the `DurableHttpRequest` and `DurableHttpResponse` types instead of the built-in .NET `HttpRequestMessage` and `HttpResponseMessage` types.

This design choice is intentional. The primary reason is that custom types help ensure users don't make incorrect assumptions about the supported behaviors of the internal HTTP client. Types specific to Durable Functions also make it possible to simplify API design. They also can more easily make available special features like [managed identity integration](#) and the [polling consumer pattern](#).

Extensibility (.NET only)

Customizing the behavior of the orchestration's internal HTTP client is possible using [Azure Functions .NET dependency injection](#). This ability can be useful for making small behavioral changes. It can also be useful for unit testing the HTTP client by injecting mock objects.

The following example demonstrates using dependency injection to disable TLS/SSL certificate validation for orchestrator functions that call external HTTP endpoints.

```
public class Startup : FunctionsStartup
{
    public override void Configure(IFunctionsHostBuilder builder)
    {
        // Register own factory
        builder.Services.AddSingleton<
            IDurableHttpMessageHandlerFactory,
            MyDurableHttpMessageHandlerFactory>();
    }
}

public class MyDurableHttpMessageHandlerFactory : IDurableHttpMessageHandlerFactory
{
    public HttpMessageHandler CreateHttpMessageHandler()
    {
        // Disable TLS/SSL certificate validation (not recommended in production!)
        return new HttpClientHandler
        {
            ServerCertificateCustomValidationCallback =
                HttpClientHandler.DangerousAcceptAnyServerCertificateValidator,
        };
    }
}
```

Next steps

[Learn about durable entities](#)

Entity functions

10/5/2022 • 15 minutes to read • [Edit Online](#)

Entity functions define operations for reading and updating small pieces of state, known as *durable entities*. Like orchestrator functions, entity functions are functions with a special trigger type, the *entity trigger*. Unlike orchestrator functions, entity functions manage the state of an entity explicitly, rather than implicitly representing state via control flow. Entities provide a means for scaling out applications by distributing the work across many entities, each with a modestly sized state.

NOTE

Entity functions and related functionality are only available in [Durable Functions 2.0](#) and above. They are currently supported in .NET, JavaScript, and Python, but not in PowerShell or Java.

General concepts

Entities behave a bit like tiny services that communicate via messages. Each entity has a unique identity and an internal state (if it exists). Like services or objects, entities perform operations when prompted to do so. When an operation executes, it might update the internal state of the entity. It might also call external services and wait for a response. Entities communicate with other entities, orchestrations, and clients by using messages that are implicitly sent via reliable queues.

To prevent conflicts, all operations on a single entity are guaranteed to execute serially, that is, one after another.

NOTE

When an entity is invoked, it processes its payload to completion and then schedules a new execution to activate once future inputs arrive. As a result, your entity execution logs might show an extra execution after each entity invocation; this is expected.

Entity ID

Entities are accessed via a unique identifier, the *entity ID*. An entity ID is simply a pair of strings that uniquely identifies an entity instance. It consists of an:

- **Entity name**, which is a name that identifies the type of the entity. An example is "Counter." This name must match the name of the entity function that implements the entity. It isn't sensitive to case.
- **Entity key**, which is a string that uniquely identifies the entity among all other entities of the same name. An example is a GUID.

For example, a `Counter` entity function might be used for keeping score in an online game. Each instance of the game has a unique entity ID, such as `@Counter@Game1` and `@Counter@Game2`. All operations that target a particular entity require specifying an entity ID as a parameter.

Entity operations

To invoke an operation on an entity, specify the:

- **Entity ID** of the target entity.
- **Operation name**, which is a string that specifies the operation to perform. For example, the `Counter` entity could support `add`, `get`, or `reset` operations.

- **Operation input**, which is an optional input parameter for the operation. For example, the add operation can take an integer amount as the input.
- **Scheduled time**, which is an optional parameter for specifying the delivery time of the operation. For example, an operation can be reliably scheduled to run several days in the future.

Operations can return a result value or an error result, such as a JavaScript error or a .NET exception. This result or error can be observed by orchestrations that called the operation.

An entity operation can also create, read, update, and delete the state of the entity. The state of the entity is always durably persisted in storage.

Define entities

Currently, the two distinct APIs for defining entities are:

Function-based syntax, where entities are represented as functions and operations are explicitly dispatched by the application. This syntax works well for entities with simple state, few operations, or a dynamic set of operations like in application frameworks. This syntax can be tedious to maintain because it doesn't catch type errors at compile time.

Class-based syntax (.NET only), where entities and operations are represented by classes and methods. This syntax produces more easily readable code and allows operations to be invoked in a type-safe way. The class-based syntax is a thin layer on top of the function-based syntax, so both variants can be used interchangeably in the same application.

- [C#](#)
- [JavaScript](#)
- [Python](#)

Example: Function-based syntax - C#

The following code is an example of a simple `Counter` entity implemented as a durable function. This function defines three operations, `add`, `reset`, and `get`, each of which operates on an integer state.

```
[FunctionName("Counter")]
public static void Counter([EntityTrigger] IDurableEntityContext ctx)
{
    switch (ctx.OperationName.ToLowerInvariant())
    {
        case "add":
            ctx.SetState(ctx.GetState<int>() + ctx.GetInput<int>());
            break;
        case "reset":
            ctx.SetState(0);
            break;
        case "get":
            ctx.Return(ctx.GetState<int>());
            break;
    }
}
```

For more information on the function-based syntax and how to use it, see [Function-based syntax](#).

Example: Class-based syntax - C#

The following example is an equivalent implementation of the `Counter` entity using classes and methods.

```
[JsonObject(MemberSerialization.OptIn)]
public class Counter
{
    [JsonProperty("value")]
    public int CurrentValue { get; set; }

    public void Add(int amount) => this.CurrentValue += amount;

    public void Reset() => this.CurrentValue = 0;

    public int Get() => this.CurrentValue;

    [FunctionName(nameof(Counter))]
    public static Task Run([EntityTrigger] IDurableEntityContext ctx)
        => ctx.DispatchAsync<Counter>();
}
```

The state of this entity is an object of type `Counter`, which contains a field that stores the current value of the counter. To persist this object in storage, it's serialized and deserialized by the [Json.NET library](#).

For more information on the class-based syntax and how to use it, see [Defining entity classes](#).

Access entities

Entities can be accessed using one-way or two-way communication. The following terminology distinguishes the two forms of communication:

- **Calling** an entity uses two-way (round-trip) communication. You send an operation message to the entity, and then wait for the response message before you continue. The response message can provide a result value or an error result, such as a JavaScript error or a .NET exception. This result or error is then observed by the caller.
- **Signaling** an entity uses one-way (fire and forget) communication. You send an operation message but don't wait for a response. While the message is guaranteed to be delivered eventually, the sender doesn't know when and can't observe any result value or errors.

Entities can be accessed from within client functions, from within orchestrator functions, or from within entity functions. Not all forms of communication are supported by all contexts:

- From within clients, you can signal entities and you can read the entity state.
- From within orchestrations, you can signal entities and you can call entities.
- From within entities, you can signal entities.

The following examples illustrate these various ways of accessing entities.

Example: Client signals an entity

To access entities from an ordinary Azure Function, which is also known as a client function, use the [entity client binding](#). The following example shows a queue-triggered function signaling an entity using this binding.

- [C#](#)
- [JavaScript](#)
- [Python](#)

NOTE

For simplicity, the following examples show the loosely typed syntax for accessing entities. In general, we recommend that you [access entities through interfaces](#) because it provides more type checking.

```
[FunctionName("AddFromQueue")]
public static Task Run(
    [QueueTrigger("durable-function-trigger")] string input,
    [DurableClient] IDurableEntityClient client)
{
    // Entity operation input comes from the queue message content.
    var entityId = new EntityId(nameof(Counter), "myCounter");
    int amount = int.Parse(input);
    return client.SignalEntityAsync(entityId, "Add", amount);
}
```

The term *signal* means that the entity API invocation is one-way and asynchronous. It's not possible for a client function to know when the entity has processed the operation. Also, the client function can't observe any result values or exceptions.

Example: Client reads an entity state

Client functions can also query the state of an entity, as shown in the following example:

- [C#](#)
- [JavaScript](#)
- [Python](#)

```
[FunctionName("QueryCounter")]
public static async Task<HttpResponseMessage> Run(
    [HttpTrigger(AuthorizationLevel.Function)] HttpRequestMessage req,
    [DurableClient] IDurableEntityClient client)
{
    var entityId = new EntityId(nameof(Counter), "myCounter");
    EntityStateResponse<JObject> stateResponse = await client.ReadEntityStateAsync<JObject>(entityId);
    return req.CreateResponse(HttpStatusCode.OK, stateResponse.EntityState);
}
```

Entity state queries are sent to the Durable tracking store and return the entity's most recently persisted state. This state is always a "committed" state, that is, it's never a temporary intermediate state assumed in the middle of executing an operation. However, it's possible that this state is stale compared to the entity's in-memory state. Only orchestrations can read an entity's in-memory state, as described in the following section.

Example: Orchestration signals and calls an entity

Orchestrator functions can access entities by using APIs on the [orchestration trigger binding](#). The following example code shows an orchestrator function calling and signaling a `Counter` entity.

- [C#](#)
- [JavaScript](#)
- [Python](#)

```
[FunctionName("CounterOrchestration")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    var entityId = new EntityId(nameof(Counter), "myCounter");

    // Two-way call to the entity which returns a value - awaits the response
    int currentValue = await context.CallEntityAsync<int>(entityId, "Get");
    if (currentValue < 10)
    {
        // One-way signal to the entity which updates the value - does not await a response
        context.SignalEntity(entityId, "Add", 1);
    }
}
```

Only orchestrations are capable of calling entities and getting a response, which could be either a return value or an exception. Client functions that use the [client binding](#) can only signal entities.

NOTE

Calling an entity from an orchestrator function is similar to calling an [activity function](#) from an orchestrator function. The main difference is that entity functions are durable objects with an address, which is the entity ID. Entity functions support specifying an operation name. Activity functions, on the other hand, are stateless and don't have the concept of operations.

Example: Entity signals an entity

An entity function can send signals to other entities, or even itself, while it executes an operation. For example, we can modify the previous `Counter` entity example so that it sends a "milestone-reached" signal to some monitor entity when the counter reaches the value 100.

- [C#](#)
- [JavaScript](#)
- [Python](#)

```
case "add":
    var currentValue = ctx.GetState<int>();
    var amount = ctx.GetInput<int>();
    if (currentValue < 100 && currentValue + amount >= 100)
    {
        ctx.SignalEntity(new EntityId("MonitorEntity", ""), "milestone-reached", ctx.EntityKey);
    }

    ctx.SetState(currentValue + amount);
    break;
```

Entity coordination (currently .NET only)

There might be times when you need to coordinate operations across multiple entities. For example, in a banking application, you might have entities that represent individual bank accounts. When you transfer funds from one account to another, you must ensure that the source account has sufficient funds. You also must ensure that updates to both the source and destination accounts are done in a transactionally consistent way.

Example: Transfer funds (C#)

The following example code transfers funds between two account entities by using an orchestrator function. Coordinating entity updates requires using the `LockAsync` method to create a *critical section* in the

orchestration.

NOTE

For simplicity, this example reuses the `Counter` entity defined previously. In a real application, it would be better to define a more detailed `BankAccount` entity.

```
// This is a method called by an orchestrator function
public static async Task<bool> TransferFundsAsync(
    string sourceId,
    string destinationId,
    int transferAmount,
    IDurableOrchestrationContext context)
{
    var sourceEntity = new EntityId(nameof(Counter), sourceId);
    var destinationEntity = new EntityId(nameof(Counter), destinationId);

    // Create a critical section to avoid race conditions.
    // No operations can be performed on either the source or
    // destination accounts until the locks are released.
    using (await context.LockAsync(sourceEntity, destinationEntity))
    {
        ICounter sourceProxy =
            context.CreateEntityProxy<ICounter>(sourceEntity);
        ICounter destinationProxy =
            context.CreateEntityProxy<ICounter>(destinationEntity);

        int sourceBalance = await sourceProxy.Get();

        if (sourceBalance >= transferAmount)
        {
            await sourceProxy.Add(-transferAmount);
            await destinationProxy.Add(transferAmount);

            // the transfer succeeded
            return true;
        }
        else
        {
            // the transfer failed due to insufficient funds
            return false;
        }
    }
}
```

In .NET, `LockAsync` returns `IDisposable`, which ends the critical section when disposed. This `IDisposable` result can be used together with a `using` block to get a syntactic representation of the critical section.

In the preceding example, an orchestrator function transferred funds from a source entity to a destination entity. The `LockAsync` method locked both the source and destination account entities. This locking ensured that no other client could query or modify the state of either account until the orchestration logic exited the critical section at the end of the `using` statement. This behavior prevents the possibility of overdrafting from the source account.

NOTE

When an orchestration terminates, either normally or with an error, any critical sections in progress are implicitly ended and all locks are released.

Critical section behavior

The `LockAsync` method creates a critical section in an orchestration. These critical sections prevent other orchestrations from making overlapping changes to a specified set of entities. Internally, the `LockAsync` API sends "lock" operations to the entities and returns when it receives a "lock acquired" response message from each of these same entities. Both lock and unlock are built-in operations supported by all entities.

No operations from other clients are allowed on an entity while it's in a locked state. This behavior ensures that only one orchestration instance can lock an entity at a time. If a caller tries to invoke an operation on an entity while it's locked by an orchestration, that operation is placed in a pending operation queue. No pending operations are processed until after the holding orchestration releases its lock.

NOTE

This behavior is slightly different from synchronization primitives used in most programming languages, such as the `lock` statement in C#. For example, in C#, the `lock` statement must be used by all threads to ensure proper synchronization across multiple threads. Entities, however, don't require all callers to explicitly lock an entity. If any caller locks an entity, all other operations on that entity are blocked and queued behind that lock.

Locks on entities are durable, so they persist even if the executing process is recycled. Locks are internally persisted as part of an entity's durable state.

Unlike transactions, critical sections don't automatically roll back changes in the case of errors. Instead, any error handling, such as roll-back or retry, must be explicitly coded, for example by catching errors or exceptions. This design choice is intentional. Automatically rolling back all the effects of an orchestration is difficult or impossible in general, because orchestrations might run activities and make calls to external services that can't be rolled back. Also, attempts to roll back might themselves fail and require further error handling.

Critical section rules

Unlike low-level locking primitives in most programming languages, critical sections are *guaranteed not to deadlock*. To prevent deadlocks, we enforce the following restrictions:

- Critical sections can't be nested.
- Critical sections can't create suborchestrations.
- Critical sections can call only entities they have locked.
- Critical sections can't call the same entity using multiple parallel calls.
- Critical sections can signal only entities they haven't locked.

Any violations of these rules cause a runtime error, such as `LockingRulesViolationException` in .NET, which includes a message that explains what rule was broken.

Comparison with virtual actors

Many of the durable entities features are inspired by the [actor model](#). If you're already familiar with actors, you might recognize many of the concepts described in this article. Durable entities are particularly similar to [virtual actors](#), or grains, as popularized by the [Orleans project](#). For example:

- Durable entities are addressable via an entity ID.
- Durable entity operations execute serially, one at a time, to prevent race conditions.
- Durable entities are created implicitly when they're called or signaled.
- When not executing operations, durable entities are silently unloaded from memory.

There are some important differences that are worth noting:

- Durable entities prioritize durability over latency, and so might not be appropriate for applications with strict latency requirements.

- Durable entities don't have built-in timeouts for messages. In Orleans, all messages time out after a configurable time. The default is 30 seconds.
- Messages sent between entities are delivered reliably and in order. In Orleans, reliable or ordered delivery is supported for content sent through streams, but isn't guaranteed for all messages between grains.
- Request-response patterns in entities are limited to orchestrations. From within entities, only one-way messaging (also known as signaling) is permitted, as in the original actor model, and unlike grains in Orleans.
- Durable entities don't deadlock. In Orleans, deadlocks can occur and don't resolve until messages time out.
- Durable entities can be used in conjunction with durable orchestrations and support distributed locking mechanisms.

Next steps

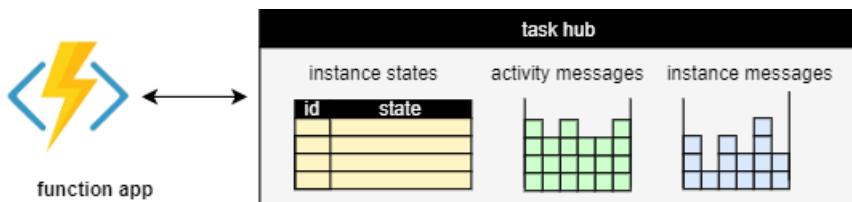
[Read the Developer's guide to durable entities in .NET](#)

[Learn about task hubs](#)

Task hubs in Durable Functions (Azure Functions)

10/5/2022 • 13 minutes to read • [Edit Online](#)

A **task hub** in [Durable Functions](#) is a representation of the current state of the application in storage, including all the pending work. While a function app is running, the progress of orchestration, activity, and entity functions is continually stored in the task hub. This ensures that the application can resume processing where it left off, should it require to be restarted after being temporarily stopped or interrupted for some reason. Also, it allows the function app to scale the compute workers dynamically.



Conceptually, a task hub stores the following information:

- The **instance states** of all orchestration and entity instances.
- The messages to be processed, including
 - any **activity messages** that represent activities waiting to be run.
 - any **instance messages** that are waiting to be delivered to instances.

The difference between activity and instance messages is that activity messages are stateless, and can thus be processed anywhere, while instance messages need to be delivered to a particular stateful instance (orchestration or entity), identified by its instance ID.

Internally, each storage provider may use a different organization to represent instance states and messages. For example, messages are stored in Azure Storage Queues by the Azure Storage provider, but in relational tables by the MSSQL provider. These differences don't matter as far as the design of the application is concerned, but some of them may influence the performance characteristics. We discuss them in the section [Representation in storage](#) below.

Work items

The activity messages and instance messages in the task hub represent the work that the function app needs to process. While the function app is running, it continuously fetches *work items* from the task hub. Each work item is processing one or more messages. We distinguish two types of work items:

- **Activity work items:** Run an activity function to process an activity message.
- **Orchestrator work item:** Run an orchestrator or entity function to process one or more instance messages.

Workers can process multiple work items at the same time, subject to the [configured per-worker concurrency limits](#).

Once a worker completes a work item, it commits the effects back to the task hub. These effects vary by the type of function that was executed:

- A completed activity function creates an instance message containing the result, addressed to the parent orchestrator instance.
- A completed orchestrator function updates the orchestration state and history, and may create new messages.

- A completed entity function updates the entity state, and may also create new instance messages.

For orchestrations, each work item represents one **episode** of that orchestration's execution. An episode starts when there are new messages for the orchestrator to process. Such a message may indicate that the orchestration should start; or it may indicate that an activity, entity call, timer, or suborchestration has completed; or it can represent an external event. The message triggers a work item that allows the orchestrator to process the result and to continue with the next episode. That episode ends when the orchestrator either completes, or reaches a point where it must wait for new messages.

Execution example

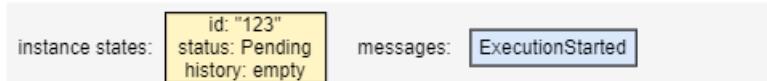
Consider a fan-out-fan-in orchestration that starts two activities in parallel, and waits for both of them to complete:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

```
[FunctionName("Example")]
public static async Task Run([OrchestrationTrigger] IDurableOrchestrationContext context)
{
    Task t1 = context.CallActivityAsync<int>("MyActivity", 1);
    Task t2 = context.CallActivityAsync<int>("MyActivity", 2);
    await Task.WhenAll(t1, t2);
}
```

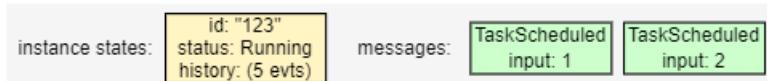
After this orchestration is initiated by a client it's processed by the function app as a sequence of work items. Each completed work item updates the task hub state when it commits. These are the steps:

1. A client requests to start a new orchestration with instance-id "123". After the client completes this request, the task hub contains a placeholder for the orchestration state and an instance message:



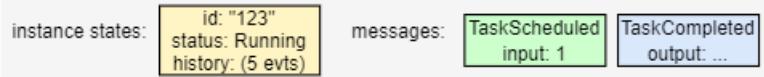
The label `ExecutionStarted` is one of many [history event types](#) that identify the various types of messages and events participating in an orchestration's history.

2. A worker executes an *orchestrator work item* to process the `ExecutionStarted` message. It calls the orchestrator function which starts executing the orchestration code. This code schedules two activities and then stops executing when it is waiting for the results. After the worker commits this work item, the task hub contains

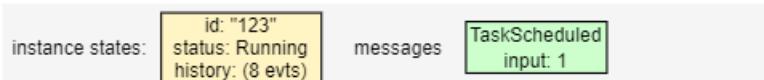


The runtime state is now `Running`, two new `TaskScheduled` messages were added, and the history now contains the five events `OrchestratorStarted`, `ExecutionStarted`, `TaskScheduled`, `TaskScheduled`, `OrchestratorCompleted`. These events represent the first episode of this orchestration's execution.

3. A worker executes an *activity work item* to process one of the `TaskScheduled` messages. It calls the activity function with input "2". When the activity function completes, it creates a `TaskCompleted` message containing the result. After the worker commits this work item, the task hub contains

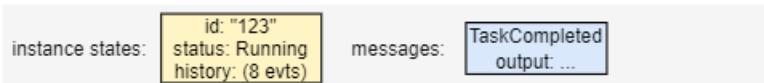


4. A worker executes an *orchestrator work item* to process the `TaskCompleted` message. If the orchestration is still cached in memory, it can just resume execution. Otherwise, the worker first [replays the history to recover the current state of the orchestration](#). Then it continues the orchestration, delivering the result of the activity. After receiving this result, the orchestration is still waiting for the result of the other activity, so it once more stops executing. After the worker commits this work item, the task hub contains

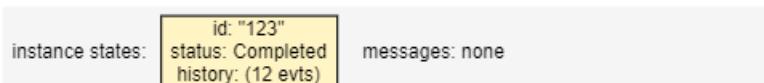


The orchestration history now contains three more events `OrchestratorStarted`, `TaskCompleted`, `OrchestratorCompleted`. These events represent the second episode of this orchestration's execution.

5. A worker executes an *activity work item* to process the remaining `TaskScheduled` message. It calls the activity function with input "1". After the worker commits this work item, the task hub contains



6. A worker executes another *orchestrator work item* to process the `TaskCompleted` message. After receiving this second result, the orchestration completes. After the worker commits this work item, the task hub contains



The runtime state is now `Completed`, and the orchestration history now contains four more events `OrchestratorStarted`, `TaskCompleted`, `ExecutionCompleted`, `OrchestratorCompleted`. These events represent the third and final episode of this orchestration's execution.

The final history for this orchestration's execution then contains the 12 events `OrchestratorStarted`, `ExecutionStarted`, `TaskScheduled`, `TaskScheduled`, `OrchestratorCompleted`, `OrchestratorStarted`, `TaskCompleted`, `OrchestratorCompleted`, `OrchestratorStarted`, `TaskCompleted`, `ExecutionCompleted`, `OrchestratorCompleted`.

NOTE

The schedule shown isn't the only one: there are many slightly different possible schedules. For example, if the second activity completes earlier, both `TaskCompleted` instance messages may be processed by a single work item. In that case, the execution history is a bit shorter, because there are only two episodes, and it contains the following 10 events:

```
OrchestratorStarted, ExecutionStarted, TaskScheduled, TaskScheduled, OrchestratorCompleted,  
OrchestratorStarted, TaskCompleted, TaskCompleted, ExecutionCompleted, OrchestratorCompleted.
```

Task hub management

Next, let's take a closer look at how task hubs are created or deleted, how to use task hubs correctly when running multiple function apps, and how the content of task hubs can be inspected.

Creation and deletion

An empty task hub with all the required resources is automatically created in storage when a function app is started the first time.

If using the default Azure Storage provider, no extra configuration is required. Otherwise, follow the [instructions](#)

for configuring storage providers to ensure that the storage provider can properly provision and access the storage resources required for the task hub.

NOTE

The task hub is *not* automatically deleted when you stop or delete the function app. You must delete the task hub, its contents, or the containing storage account manually if you no longer want to keep that data.

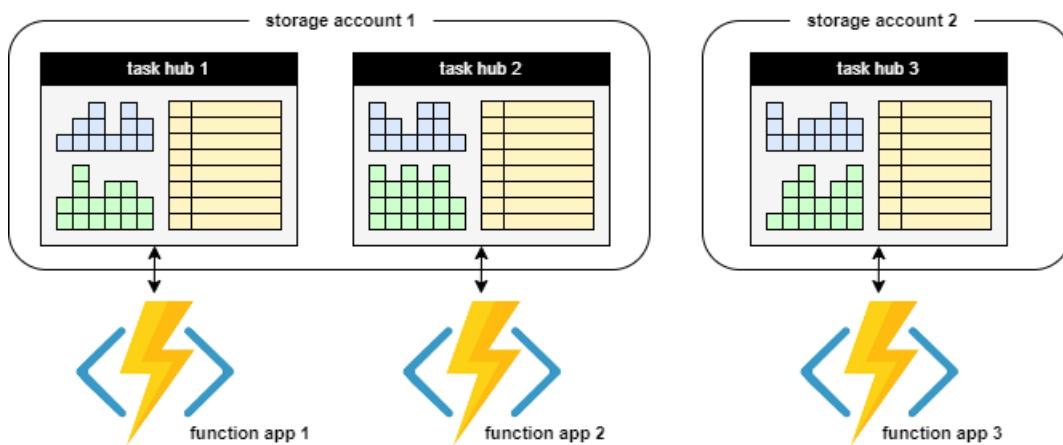
TIP

In a development scenario, you may need to restart from a clean state often. To do so quickly, you can just [change the configured task hub name](#). This will force the creation of a new, empty task hub when you restart the application. Be aware that the old data is not deleted in this case.

Multiple function apps

If multiple function apps share a storage account, each function app *must* be configured with a separate [task hub name](#). This requirement also applies to staging slots: each staging slot must be configured with a unique task hub name. A single storage account can contain multiple task hubs. This restriction generally applies to other storage providers as well.

The following diagram illustrates one task hub per function app in shared and dedicated Azure Storage accounts.



NOTE

The exception to the task hub sharing rule is if you are configuring your app for regional disaster recovery. See the [disaster recovery and geo-distribution](#) article for more information.

Content inspection

There are several common ways to inspect the contents of a task hub:

1. Within a function app, the client object provides methods to query the instance store. To learn more about what types of queries are supported, see the [Instance Management](#) article.
2. Similarly, The [HTTP API](#) offers REST requests to query the state of orchestrations and entities. See the [HTTP API Reference](#) for more details.
3. The [Durable Functions Monitor](#) tool can inspect task hubs and offers various options for visual display.

For some of the storage providers, it is also possible to inspect the task hub by going directly to the underlying storage:

- If using the Azure Storage provider, the instance states are stored in the [Instance Table](#) and the [History Table](#)

that can be inspected using tools such as Azure Storage Explorer.

- If using the MSSQL storage provider, SQL queries and tools can be used to inspect the task hub contents inside the database.

Representation in storage

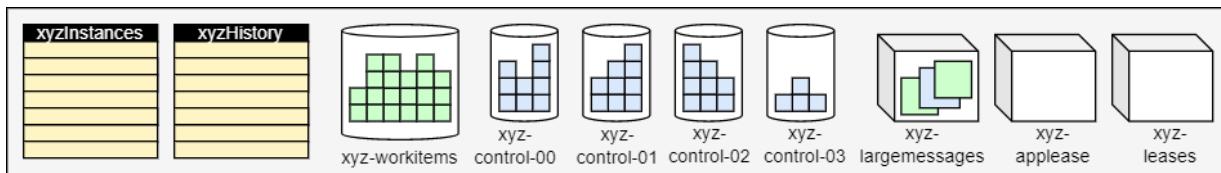
Each storage provider uses a different internal organization to represent task hubs in storage. Understanding this organization, while not required, can be helpful when troubleshooting a function app or when trying to ensure performance, scalability, or cost targets. We thus briefly explain, for each storage provider, how the data is organized in storage. For more information on the various storage provider options and how they compare, see the [Durable Functions storage providers](#).

Azure Storage provider

The Azure Storage provider represents the task hub in storage using the following components:

- Two Azure Tables store the instance states.
- One Azure Queue stores the activity messages.
- One or more Azure Queues store the instance messages. Each of these so-called *control queues* represents a *partition* that is assigned a subset of all instance messages, based on the hash of the instance ID.
- A few extra blob containers used for lease blobs and/or large messages.

For example, a task hub named `xyz` with `PartitionCount = 4` contains the following queues and tables:



Next, we describe these components and the role they play in more detail.

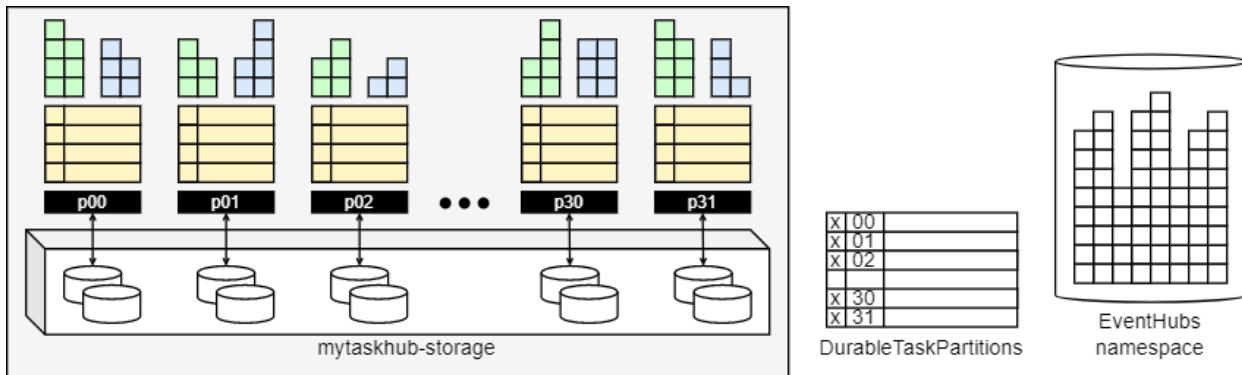
For more information how task hubs are represented by the Azure Storage provider, see the [Azure Storage provider](#) documentation.

Netherite storage provider

Netherite partitions all of the task hub state into a specified number of partitions. In storage, the following resources are used:

- One Azure Storage blob container that contains all the blobs, grouped by partition.
- One Azure Table that contains published metrics about the partitions.
- An Azure Event Hubs namespace for delivering messages between partitions.

For example, a task hub named `mytaskhub` with `PartitionCount = 32` is represented in storage as follows:



NOTE

All of the task hub state is stored inside the `x-storage` blob container. The `DurableTaskPartitions` table and the EventHubs namespace contain redundant data: if their contents are lost, they can be automatically recovered. Therefore it is not necessary to configure the Azure Event Hubs namespace to retain messages past the default expiration time.

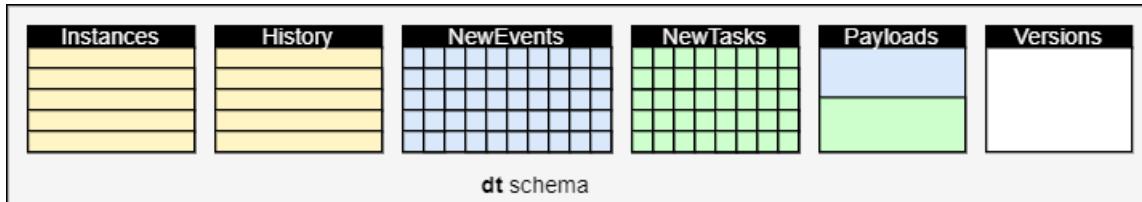
Netherite uses an event-sourcing mechanism, based on a log and checkpoints, to represent the current state of a partition. Both block blobs and page blobs are used. It is not possible to read this format from storage directly, so the function app has to be running when querying the instance store.

For more information on task hubs for the Netherite storage provider, see [Task Hub information for the Netherite storage provider](#).

MSSQL storage provider

All task hub data is stored in a single relational database, using several tables:

- The `dt.Instances` and `dt.History` tables store the instance states.
- The `dt.NewEvents` table stores the instance messages.
- The `dt.NewTasks` table stores the activity messages.



To enable multiple task hubs to coexist independently in the same database, each table includes a `TaskHub` column as part of its primary key. Unlike the other two providers, the MSSQL provider doesn't have a concept of partitions.

For more information on task hubs for the MSSQL storage provider, see [Task Hub information for the Microsoft SQL \(MSSQL\) storage provider](#).

Task hub names

Task hubs are identified by a name that must conform to these rules:

- Contains only alphanumeric characters
- Starts with a letter
- Has a minimum length of 3 characters, maximum length of 45 characters

The task hub name is declared in the `host.json` file, as shown in the following example:

host.json (Functions 2.0)

```
{  
  "version": "2.0",  
  "extensions": {  
    "durableTask": {  
      "hubName": "MyTaskHub"  
    }  
  }  
}
```

host.json (Functions 1.x)

```
{  
  "durableTask": {  
    "hubName": "MyTaskHub"  
  }  
}
```

Task hubs can also be configured using app settings, as shown in the following `host.json` example file:

host.json (Functions 1.0)

```
{  
  "durableTask": {  
    "hubName": "%MyTaskHub%"  
  }  
}
```

host.json (Functions 2.0)

```
{  
  "version": "2.0",  
  "extensions": {  
    "durableTask": {  
      "hubName": "%MyTaskHub%"  
    }  
  }  
}
```

The task hub name will be set to the value of the `MyTaskHub` app setting. The following `local.settings.json` demonstrates how to define the `MyTaskHub` setting as `samplehubname`:

```
{  
  "IsEncrypted": false,  
  "Values": {  
    "MyTaskHub" : "samplehubname"  
  }  
}
```

NOTE

When using deployment slots, it's a best practice to configure the task hub name using app settings. If you want to ensure that a particular slot always uses a particular task hub, use ["slot-sticky" app settings](#).

In addition to `host.json`, task hub names can also be configured in [orchestration client binding](#) metadata. This is useful if you need to access orchestrations or entities that live in a separate function app. The following code demonstrates how to write a function that uses the [orchestration client binding](#) to work with a task hub that is configured as an App Setting:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

```
[FunctionName("HttpStart")]
public static async Task<HttpResponseMessage> Run(
    [HttpTrigger(AuthorizationLevel.Function, methods: "post", Route = "orchestrators/{functionName}")]
    HttpRequestMessage req,
    [DurableClient(TaskHub = "%MyTaskHub%")] IDurableOrchestrationClient starter,
    string functionName,
    ILogger log)
{
    // Function input comes from the request content.
    object eventData = await req.Content.ReadAsAsync<object>();
    string instanceId = await starter.StartNewAsync(functionName, eventData);

    log.LogInformation($"Started orchestration with ID = '{instanceId}'.");

    return starter.CreateCheckStatusResponse(req, instanceId);
}
```

NOTE

The previous C# example is for Durable Functions 2.x. For Durable Functions 1.x, you must use `DurableOrchestrationContext` instead of `IDurableOrchestrationContext`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

NOTE

Configuring task hub names in client binding metadata is only necessary when you use one function app to access orchestrations and entities in another function app. If the client functions are defined in the same function app as the orchestrations and entities, you should avoid specifying task hub names in the binding metadata. By default, all client bindings get their task hub metadata from the `host.json` settings.

Task hub names must start with a letter and consist of only letters and numbers. If not specified, a default task hub name will be used as shown in the following table:

DURABLE EXTENSION VERSION	DEFAULT TASK HUB NAME
2.x	When deployed in Azure, the task hub name is derived from the name of the <i>function app</i> . When running outside of Azure, the default task hub name is <code>TestHubName</code> .
1.x	The default task hub name for all environments is <code>DurableFunctionsHub</code> .

For more information about the differences between extension versions, see the [Durable Functions versions](#) article.

NOTE

The name is what differentiates one task hub from another when there are multiple task hubs in a shared storage account. If you have multiple function apps sharing a shared storage account, you must explicitly configure different names for each task hub in the `host.json` files. Otherwise the multiple function apps will compete with each other for messages, which could result in undefined behavior, including orchestrations getting unexpectedly "stuck" in the `Pending` or `Running` state.

Next steps

[Learn how to handle orchestration versioning](#)

Manage instances in Durable Functions in Azure

10/5/2022 • 31 minutes to read • [Edit Online](#)

Orchestrations in Durable Functions are long-running stateful functions that can be started, queried, suspended, resumed, and terminated using built-in management APIs. Several other instance management APIs are also exposed by the Durable Functions [orchestration client binding](#), such as sending external events to instances, purging instance history, etc. This article goes into the details of all supported instance management operations.

Start instances

The *start-new* (or *schedule-new*) method on the [orchestration client binding](#) starts a new orchestration instance. Internally, this method writes a message via the [Durable Functions storage provider](#) and then returns. This message asynchronously triggers the start of an [orchestration function](#) with the specified name.

The parameters for starting a new orchestration instance are as follows:

- **Name:** The name of the orchestrator function to schedule.
- **Input:** Any JSON-serializable data that should be passed as the input to the orchestrator function.
- **InstanceId:** (Optional) The unique ID of the instance. If you don't specify this parameter, the method uses a random ID.

TIP

Use a random identifier for the instance ID whenever possible. Random instance IDs help ensure an equal load distribution when you're scaling orchestrator functions across multiple VMs. The proper time to use non-random instance IDs is when the ID must come from an external source, or when you're implementing the [singleton orchestrator](#) pattern.

The following code is an example function that starts a new orchestration instance:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

```
[FunctionName("HelloWorldQueueTrigger")]
public static async Task Run(
    [QueueTrigger("start-queue")] string input,
    [DurableClient] IDurableOrchestrationClient starter,
    ILogger log)
{
    string instanceId = await starter.StartNewAsync("HelloWorld", input);
    log.LogInformation($"Started orchestration with ID = '{instanceId}'.");
}
```

NOTE

The previous C# code is for Durable Functions 2.x. For Durable Functions 1.x, you must use `OrchestrationClient` attribute instead of the `DurableClient` attribute, and you must use the `DurableOrchestrationClient` parameter type instead of `IDurableOrchestrationClient`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

Azure Functions Core Tools

You can also start an instance directly by using the `func durable start-new` command in Core Tools, which takes the following parameters:

- `function-name` (**required**): Name of the function to start.
- `input` (**optional**): Input to the function, either inline or through a JSON file. For files, add a prefix to the path to the file with `@`, such as `@path/to/file.json`.
- `id` (**optional**): ID of the orchestration instance. If you don't specify this parameter, the command uses a random GUID.
- `connection-string-setting` (**optional**): Name of the application setting containing the storage connection string to use. The default is `AzureWebJobsStorage`.
- `task-hub-name` (**optional**): Name of the Durable Functions task hub to use. The default is `DurableFunctionsHub`. You can also set this in `host.json` by using `durableTask:HubName`.

NOTE

Core Tools commands assume you are running them from the root directory of a function app. If you explicitly provide the `connection-string-setting` and `task-hub-name` parameters, you can run the commands from any directory. Although you can run these commands without a function app host running, you might find that you can't observe some effects unless the host is running. For example, the `start-new` command enqueues a start message into the target task hub, but the orchestration doesn't actually run unless there is a function app host process running that can process the message.

NOTE

The Core Tools commands are currently only supported when using the default [Azure Storage provider](#) for persisting runtime state.

The following command starts the function named `HelloWorld`, and passes the contents of the file `counter-data.json` to it:

```
func durable start-new --function-name HelloWorld --input @counter-data.json --task-hub-name TestTaskHub
```

Query instances

After starting new orchestration instances, you'll most likely need to query their runtime status to learn whether they are running, have completed, or have failed.

The `get-status` method on the [orchestration client binding](#) queries the status of an orchestration instance.

It takes an `instanceId` (**required**), `showHistory` (**optional**), `showHistoryOutput` (**optional**), and `showInput` (**optional**) as parameters.

- `showHistory`: If set to `true`, the response contains the execution history.

- `showHistoryOutput` : If set to `true`, the execution history contains activity outputs.
- `showInput` : If set to `false`, the response won't contain the input of the function. The default value is `true`.

The method returns an object with the following properties:

- **Name**: The name of the orchestrator function.
- **InstanceId**: The instance ID of the orchestration (should be the same as the `instanceId` input).
- **CreatedTime**: The time at which the orchestrator function started running.
- **LastUpdatedTime**: The time at which the orchestration last checkpointed.
- **Input**: The input of the function as a JSON value. This field isn't populated if `showInput` is false.
- **CustomStatus**: Custom orchestration status in JSON format.
- **Output**: The output of the function as a JSON value (if the function has completed). If the orchestrator function failed, this property includes the failure details. If the orchestrator function was suspended or terminated, this property includes the reason for the suspension or termination (if any).
- **RuntimeStatus**: One of the following values:
 - **Pending**: The instance has been scheduled but has not yet started running.
 - **Running**: The instance has started running.
 - **Completed**: The instance has completed normally.
 - **ContinuedAsNew**: The instance has restarted itself with a new history. This state is a transient state.
 - **Failed**: The instance failed with an error.
 - **Terminated**: The instance was stopped abruptly.
 - **Suspended**: The instance was suspended and may be resumed at a later point in time.
- **History**: The execution history of the orchestration. This field is only populated if `showHistory` is set to `true`
- `.`

NOTE

An orchestrator is not marked as `Completed` until all of its scheduled tasks have finished *and* the orchestrator has returned. In other words, it is not sufficient for an orchestrator to reach its `return` statement for it to be marked as `Completed`. This is particularly relevant for cases where `WhenAny` is used; those orchestrators often `return` before all the scheduled tasks have executed.

This method returns `null` (.NET and Java), `undefined` (JavaScript), or `None` (Python) if the instance doesn't exist.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

```
[FunctionName("GetStatus")]
public static async Task Run(
    [DurableClient] IDurableOrchestrationClient client,
    [QueueTrigger("check-status-queue")] string instanceId)
{
    DurableOrchestrationStatus status = await client.GetStatusAsync(instanceId);
    // do something based on the current status.
}
```

NOTE

The previous C# code is for Durable Functions 2.x. For Durable Functions 1.x, you must use `OrchestrationClient` attribute instead of the `DurableClient` attribute, and you must use the `DurableOrchestrationClient` parameter type instead of `IDurableOrchestrationClient`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

Azure Functions Core Tools

It's also possible to get the status of an orchestration instance directly by using the

```
func durable get-runtime-status
```

NOTE

Core Tools commands are currently only supported when using the default [Azure Storage provider](#) for persisting runtime state.

The `durable get-runtime-status` command takes the following parameters:

- `id` **(required)**: ID of the orchestration instance.
- `show-input` **(optional)**: If set to `true`, the response contains the input of the function. The default value is `false`.
- `show-output` **(optional)**: If set to `true`, the response contains the output of the function. The default value is `false`.
- `connection-string-setting` **(optional)**: Name of the application setting containing the storage connection string to use. The default is `AzureWebJobsStorage`.
- `task-hub-name` **(optional)**: Name of the Durable Functions task hub to use. The default is `DurableFunctionsHub`. It can also be set in `host.json`, by using `durableTask:HubName`.

The following command retrieves the status (including input and output) of an instance with an orchestration instance ID of `0ab8c55a66644d68a3a8b220b12d209c`. It assumes that you are running the `func` command from the root directory of the function app:

```
func durable get-runtime-status --id 0ab8c55a66644d68a3a8b220b12d209c --show-input true --show-output true
```

You can use the `durable get-history` command to retrieve the history of an orchestration instance. It takes the following parameters:

- `id` **(required)**: ID of the orchestration instance.
- `connection-string-setting` **(optional)**: Name of the application setting containing the storage connection string to use. The default is `AzureWebJobsStorage`.
- `task-hub-name` **(optional)**: Name of the Durable Functions task hub to use. The default is `DurableFunctionsHub`. It can also be set in `host.json`, by using `durableTask:HubName`.

```
func durable get-history --id 0ab8c55a66644d68a3a8b220b12d209c
```

Query all instances

You can use APIs in your language SDK to query the statuses of all orchestration instances in your [task hub](#). This *"list-instances"* or *"get-status"* API returns a list of objects that represent the orchestration instances matching the query parameters.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

```
[FunctionName("GetAllStatus")]
public static async Task Run(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post")] HttpRequestMessage req,
    [DurableClient] IDurableOrchestrationClient client,
    ILogger log)
{
    var noFilter = new OrchestrationStatusQueryCondition();
    OrchestrationStatusQueryResult result = await client.ListInstancesAsync(
        noFilter,
        CancellationToken.None);
    foreach (DurableOrchestrationStatus instance in result.DurableOrchestrationState)
    {
        log.LogInformation(JsonConvert.SerializeObject(instance));
    }

    // Note: ListInstancesAsync only returns the first page of results.
    // To request additional pages provide the result.ContinuationToken
    // to the OrchestrationStatusQueryCondition's ContinuationToken property.
}
```

NOTE

The previous C# code is for Durable Functions 2.x. For Durable Functions 1.x, you must use `OrchestrationClient` attribute instead of the `DurableClient` attribute, and you must use the `DurableOrchestrationClient` parameter type instead of `IDurableOrchestrationClient`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

Azure Functions Core Tools

It's also possible to query instances directly, by using the `func durable get-instances` command in Core Tools.

NOTE

The Core Tools commands are currently only supported when using the default [Azure Storage provider](#) for persisting runtime state.

The `durable get-instances` command takes the following parameters:

- `top` **(optional)**: This command supports paging. This parameter corresponds to the number of instances retrieved per request. The default is 10.
- `continuation-token` **(optional)**: A token to indicate which page or section of instances to retrieve. Each `get-instances` execution returns a token to the next set of instances.
- `connection-string-setting` **(optional)**: Name of the application setting containing the storage connection string to use. The default is `AzureWebJobsStorage`.
- `task-hub-name` **(optional)**: Name of the Durable Functions task hub to use. The default is `DurableFunctionsHub`. It can also be set in `host.json`, by using `durableTask:HubName`.

```
func durable get-instances
```

Query instances with filters

What if you don't really need all the information that a standard instance query can provide? For example, what if you're just looking for the orchestration creation time, or the orchestration runtime status? You can narrow your query by applying filters.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

```
[FunctionName("QueryStatus")]
public static async Task Run(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post")] HttpRequestMessage req,
    [DurableClient] IDurableOrchestrationClient client,
    ILogger log)
{
    // Get the first 100 running or pending instances that were created between 7 and 1 day(s) ago
    var queryFilter = new OrchestrationStatusQueryCondition
    {
        RuntimeStatus = new[]
        {
            OrchestrationRuntimeStatus.Pending,
            OrchestrationRuntimeStatus.Running,
        },
        CreatedTimeFrom = DateTime.UtcNow.Subtract(TimeSpan.FromDays(7)),
        CreatedTimeTo = DateTime.UtcNow.Subtract(TimeSpan.FromDays(1)),
        PageSize = 100,
    };

    OrchestrationStatusQueryResult result = await client.ListInstancesAsync(
        queryFilter,
        CancellationToken.None);
    foreach (DurableOrchestrationStatus instance in result.DurableOrchestrationState)
    {
        log.LogInformation(JsonConvert.SerializeObject(instance));
    }
}
```

NOTE

The previous C# code is for Durable Functions 2.x. For Durable Functions 1.x, you must use `OrchestrationClient` attribute instead of the `DurableClient` attribute, and you must use the `DurableOrchestrationClient` parameter type instead of `IDurableOrchestrationClient`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

Azure Functions Core Tools

In the Azure Functions Core Tools, you can also use the `durable get-instances` command with filters. In addition to the aforementioned `top`, `continuation-token`, `connection-string-setting`, and `task-hub-name` parameters, you can use three filter parameters (`created-after`, `created-before`, and `runtime-status`).

NOTE

The Core Tools commands are currently only supported when using the default [Azure Storage provider](#) for persisting runtime state.

The following are the parameters for the `durable get-instances` command.

- `created-after` **(optional)**: Retrieve the instances created after this date/time (UTC). ISO 8601 formatted datetimes accepted.
- `created-before` **(optional)**: Retrieve the instances created before this date/time (UTC). ISO 8601 formatted datetimes accepted.
- `runtime-status` **(optional)**: Retrieve the instances with a particular status (for example, running or completed). Can provide multiple (space separated) statuses.
- `top` **(optional)**: Number of instances retrieved per request. The default is 10.
- `continuation-token` **(optional)**: A token to indicate which page or section of instances to retrieve. Each `get-instances` execution returns a token to the next set of instances.
- `connection-string-setting` **(optional)**: Name of the application setting containing the storage connection string to use. The default is `AzureWebJobsStorage`.
- `task-hub-name` **(optional)**: Name of the Durable Functions task hub to use. The default is `DurableFunctionsHub`. It can also be set in [host.json](#), by using `durableTask:HubName`.

If you don't provide any filters (`created-after`, `created-before`, or `runtime-status`), the command simply retrieves `top` instances, with no regard to runtime status or creation time.

```
func durable get-instances --created-after 2021-03-10T13:57:31Z --created-before 2021-03-10T23:59Z --top 15
```

Terminate instances

If you have an orchestration instance that is taking too long to run, or you just need to stop it before it completes for any reason, you can terminate it.

The two parameters for the terminate API are an *instance ID* and a *reason* string, which are written to logs and to the instance status.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

```
[FunctionName("TerminateInstance")]
public static Task Run(
    [DurableClient] IDurableOrchestrationClient client,
    [QueueTrigger("terminate-queue")] string instanceId)
{
    string reason = "Found a bug";
    return client.TerminateAsync(instanceId, reason);
}
```

NOTE

The previous C# code is for Durable Functions 2.x. For Durable Functions 1.x, you must use `OrchestrationClient` attribute instead of the `DurableClient` attribute, and you must use the `DurableOrchestrationClient` parameter type instead of `IDurableOrchestrationClient`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

A terminated instance will eventually transition into the `Terminated` state. However, this transition will not

happen immediately. Rather, the terminate operation will be queued in the task hub along with other operations for that instance. You can use the [instance query](#) APIs to know when a terminated instance has actually reached the `Terminated` state.

NOTE

Instance termination doesn't currently propagate. Activity functions and sub-orchestrations run to completion, regardless of whether you've terminated the orchestration instance that called them.

Suspend and Resume instances (preview)

Suspending an orchestration allows you to stop a running orchestration. Unlike with termination, you have the option to resume a suspended orchestrator at a later point in time.

The two parameters for the suspend API are an instance ID and a reason string, which are written to logs and to the instance status.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

```
[FunctionName("SuspendResumeInstance")]
public static async Task Run(
    [DurableClient] IDurableOrchestrationClient client,
    [QueueTrigger("suspend-resume-queue")] string instanceId)
{
    string suspendReason = "Need to pause workflow";
    await client.SuspendAsync(instanceId, suspendReason);

    // ... wait for some period of time since suspending is an async operation...

    string resumeReason = "Continue workflow";
    await client.ResumeAsync(instanceId, resumeReason);
}
```

A suspended instance will eventually transition to the `Suspended` state. However, this transition will not happen immediately. Rather, the suspend operation will be queued in the task hub along with other operations for that instance. You can use the instance query APIs to know when a running instance has actually reached the `Suspended` state.

When a suspended orchestrator is resumed, its status will change back to `Running`.

Azure Functions Core Tools

You can also terminate an orchestration instance directly, by using the `func durable terminate` command in Core Tools.

NOTE

The Core Tools commands are currently only supported when using the default [Azure Storage provider](#) for persisting runtime state.

The `durable terminate` command takes the following parameters:

- `id` (required): ID of the orchestration instance to terminate.

- `reason` (**optional**): Reason for termination.
- `connection-string-setting` (**optional**): Name of the application setting containing the storage connection string to use. The default is `AzureWebJobsStorage`.
- `task-hub-name` (**optional**): Name of the Durable Functions task hub to use. The default is `DurableFunctionsHub`. It can also be set in `host.json`, by using `durableTask:HubName`.

The following command terminates an orchestration instance with an ID of `0ab8c55a66644d68a3a8b220b12d209c`:

```
func durable terminate --id 0ab8c55a66644d68a3a8b220b12d209c --reason "Found a bug"
```

Send events to instances

In some scenarios, orchestrator functions need to wait and listen for external events. Examples scenarios where this is useful include the [monitoring](#) and [human interaction](#) scenarios.

You can send event notifications to running instances by using the *raise event* API of the [orchestration client](#). Orchestrations can listen and respond to these events using the *wait for external event* orchestrator API.

The parameters for *raise event* are as follows:

- *Instance ID*: The unique ID of the instance.
 - *Event name*: The name of the event to send.
 - *Event data*: A JSON-serializable payload to send to the instance.
- [C#](#)
 - [JavaScript](#)
 - [Python](#)
 - [Java](#)

```
[FunctionName("RaiseEvent")]
public static Task Run(
    [DurableClient] IDurableOrchestrationClient client,
    [QueueTrigger("event-queue")] string instanceId)
{
    int[] eventData = new int[] { 1, 2, 3 };
    return client.RaiseEventAsync(instanceId, "MyEvent", eventData);
}
```

NOTE

The previous C# code is for Durable Functions 2.x. For Durable Functions 1.x, you must use `OrchestrationClient` attribute instead of the `DurableClient` attribute, and you must use the `DurableOrchestrationClient` parameter type instead of `IDurableOrchestrationClient`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

NOTE

If there is no orchestration instance with the specified instance ID, the event message is discarded. If an instance exists but it is not yet waiting for the event, the event will be stored in the instance state until it is ready to be received and processed.

Azure Functions Core Tools

You can also raise an event to an orchestration instance directly, by using the `func durable raise-event` command in Core Tools.

NOTE

The Core Tools commands are currently only supported when using the default [Azure Storage provider](#) for persisting runtime state.

The `durable raise-event` command takes the following parameters:

- `id` (**required**): ID of the orchestration instance.
- `event-name`: Name of the event to raise.
- `event-data` (**optional**): Data to send to the orchestration instance. This can be the path to a JSON file, or you can provide the data directly on the command line.
- `connection-string-setting` (**optional**): Name of the application setting containing the storage connection string to use. The default is `AzureWebJobsStorage`.
- `task-hub-name` (**optional**): Name of the Durable Functions task hub to use. The default is `DurableFunctionsHub`. It can also be set in [host.json](#), by using `durableTask:HubName`.

```
func durable raise-event --id 0ab8c55a66644d68a3a8b220b12d209c --event-name MyEvent --event-data @ eventdata.json
```

```
func durable raise-event --id 1234567 --event-name MyOtherEvent --event-data 3
```

Wait for orchestration completion

In long-running orchestrations, you may want to wait and get the results of an orchestration. In these cases, it's also useful to be able to define a timeout period on the orchestration. If the timeout is exceeded, the state of the orchestration should be returned instead of the results.

The "*wait for completion or create check status response*" API can be used to get the actual output from an orchestration instance synchronously. By default, this method has a default timeout of 10 seconds and a polling interval of 1 second.

Here is an example HTTP-trigger function that demonstrates how to use this API:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

```

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

using System;
using System.Net.Http;
using System.Threading.Tasks;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.DurableTask;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Extensions.Logging;

namespace VSSample
{
    public static class HttpSyncStart
    {
        private const string Timeout = "timeout";
        private const string RetryInterval = "retryInterval";

        [FunctionName("HttpSyncStart")]
        public static async Task<HttpResponseMessage> Run(
            [HttpTrigger(AuthorizationLevel.Function, methods: "post", Route =
"orchestrators/{functionName}/wait")]
            HttpRequestMessage req,
            [DurableClient] IDurableOrchestrationClient starter,
            string functionName,
            ILogger log)
        {
            // Function input comes from the request content.
            object eventData = await req.Content.ReadAsAsync<object>();
            string instanceId = await starter.StartNewAsync(functionName, eventData);

            log.LogInformation($"Started orchestration with ID = '{instanceId}'.");

            TimeSpan timeout = GetTimeSpan(req, Timeout) ?? TimeSpan.FromSeconds(30);
            TimeSpan retryInterval = GetTimeSpan(req, RetryInterval) ?? TimeSpan.FromSeconds(1);

            return await starter.WaitForCompletionOrCreateCheckStatusResponseAsync(
                req,
                instanceId,
                timeout,
                retryInterval);
        }

        private static TimeSpan? GetTimeSpan(HttpRequestMessage request, string queryParameterName)
        {
            string queryParameterStringValue = request.RequestUri.ParseQueryString()[queryParameterName];
            if (string.IsNullOrEmpty(queryParameterStringValue))
            {
                return null;
            }

            return TimeSpan.FromSeconds(double.Parse(queryParameterStringValue));
        }
    }
}

```

Call the function with the following line. Use 2 seconds for the timeout and 0.5 second for the retry interval:

```
curl -X POST "http://localhost:7071/orchestrators/E1_HelloSequence/wait?timeout=2&retryInterval=0.5"
```

NOTE

The above cURL command assumes you have an orchestrator function named `E1_HelloSequence` in your project. Because of how the HTTP trigger function is written, you can replace it with the name of any orchestrator function in your project.

Depending on the time required to get the response from the orchestration instance, there are two cases:

- The orchestration instances complete within the defined timeout (in this case 2 seconds), and the response is the actual orchestration instance output, delivered synchronously:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Thu, 14 Dec 2021 06:14:29 GMT
Transfer-Encoding: chunked

[
    "Hello Tokyo!",
    "Hello Seattle!",
    "Hello London!"
]
```

- The orchestration instances can't complete within the defined timeout, and the response is the default one described in [HTTP API URL discovery](#):

```
HTTP/1.1 202 Accepted
Content-Type: application/json; charset=utf-8
Date: Thu, 14 Dec 2021 06:13:51 GMT
Location: http://localhost:7071/runtime/webhooks/durabletask/instances/d3b72dddefce4e758d92f4d411567177?taskHub={taskHub}&connection={connection}&code={systemKey}
Retry-After: 10
Transfer-Encoding: chunked

{
    "id": "d3b72dddefce4e758d92f4d411567177",
    "sendEventPostUri":
        "http://localhost:7071/runtime/webhooks/durabletask/instances/d3b72dddefce4e758d92f4d411567177/raiseEvent/{eventName}?taskHub={taskHub}&connection={connection}&code={systemKey}",
    "statusQueryGetUri":
        "http://localhost:7071/runtime/webhooks/durabletask/instances/d3b72dddefce4e758d92f4d411567177?taskHub={taskHub}&connection={connection}&code={systemKey}",
    "terminatePostUri":
        "http://localhost:7071/runtime/webhooks/durabletask/instances/d3b72dddefce4e758d92f4d411567177/terminate?reason={text}&taskHub={taskHub}&connection={connection}&code={systemKey}",
    "suspendPostUri":
        "http://localhost:7071/runtime/webhooks/durabletask/instances/d3b72dddefce4e758d92f4d411567177/suspend?reason={text}&taskHub={taskHub}&connection={connection}&code={systemKey}",
    "resumePostUri":
        "http://localhost:7071/runtime/webhooks/durabletask/instances/d3b72dddefce4e758d92f4d411567177/resume?reason={text}&taskHub={taskHub}&connection={connection}&code={systemKey}"
}
```

NOTE

The format of the webhook URLs might differ, depending on which version of the Azure Functions host you are running. The preceding example is for the Azure Functions 3.0 host.

Retrieve HTTP management webhook URLs

You can use an external system to monitor or to raise events to an orchestration. External systems can communicate with Durable Functions through the webhook URLs that are part of the default response described in [HTTP API URL discovery](#). The webhook URLs can alternatively be accessed programmatically using the [orchestration client binding](#). Specifically, the *create HTTP management payload* API can be used to get a serializable object that contains these webhook URLs.

The *create HTTP management payload* API has one parameter:

- *InstanceId*: The unique ID of the instance.

The methods return an object with the following string properties:

- **Id**: The instance ID of the orchestration (should be the same as the `InstanceId` input).
- **StatusQueryGetUri**: The status URL of the orchestration instance.
- **SendEventPostUri**: The "raise event" URL of the orchestration instance.
- **TerminatePostUri**: The "terminate" URL of the orchestration instance.
- **PurgeHistoryDeleteUri**: The "purge history" URL of the orchestration instance.
- **suspendPostUri**: The "suspend" URL of the orchestration instance.
- **resumePostUri**: The "resume" URL of the orchestration instance.

Functions can send instances of these objects to external systems to monitor or raise events on the corresponding orchestrations, as shown in the following examples:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

```
[FunctionName("SendInstanceInfo")]
public static void SendInstanceInfo(
    [ActivityTrigger] IDurableActivityContext ctx,
    [DurableClient] IDurableOrchestrationClient client,
    [DocumentDB(
        databaseName: "MonitorDB",
        collectionName: "HttpManagementPayloads",
        ConnectionStringSetting = "CosmosDBConnection")]out dynamic document)
{
    HttpManagementPayload payload = client.CreateHttpManagementPayload(ctx.InstanceId);

    // send the payload to Cosmos DB
    document = new { Payload = payload, id = ctx.InstanceId };
}
```

NOTE

The previous C# code is for Durable Functions 2.x. For Durable Functions 1.x, you must use `DurableActivityContext` instead of `IDurableActivityContext`, you must use the `OrchestrationClient` attribute instead of the `DurableClient` attribute, and you must use the `DurableOrchestrationClient` parameter type instead of `IDurableOrchestrationClient`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

Rewind instances (preview)

If you have an orchestration failure for an unexpected reason, you can *rewind* the instance to a previously healthy state by using an API built for that purpose.

NOTE

This API is not intended to be a replacement for proper error handling and retry policies. Rather, it is intended to be used only in cases where orchestration instances fail for unexpected reasons. For more information on error handling and retry policies, see the [Error handling](#) article.

Use the `RewindAsync` (.NET) or `rewind` (JavaScript) method of the [orchestration client binding](#) to put the orchestration back into the *Running* state. This method will also rerun the activity or sub-orchestration execution failures that caused the orchestration failure.

For example, let's say you have a workflow involving a series of [human approvals](#). Suppose there are a series of activity functions that notify someone that their approval is needed, and wait out the real-time response. After all of the approval activities have received responses or timed out, suppose that another activity fails due to an application misconfiguration, such as an invalid database connection string. The result is an orchestration failure deep into the workflow. With the `RewindAsync` (.NET) or `rewind` (JavaScript) API, an application administrator can fix the configuration error, and rewind the failed orchestration back to the state immediately before the failure. None of the human-interaction steps need to be re-approved, and the orchestration can now complete successfully.

NOTE

The `rewind` feature doesn't support rewinding orchestration instances that use durable timers.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

```
[FunctionName("RewindInstance")]
public static Task Run(
    [DurableClient] IDurableOrchestrationClient client,
    [QueueTrigger("rewind-queue")] string instanceId)
{
    string reason = "Orchestrator failed and needs to be revived.";
    return client.RewindAsync(instanceId, reason);
}
```

NOTE

The previous C# code is for Durable Functions 2.x. For Durable Functions 1.x, you must use `OrchestrationClient` attribute instead of the `DurableClient` attribute, and you must use the `DurableOrchestrationClient` parameter type instead of `IDurableOrchestrationClient`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

Azure Functions Core Tools

You can also rewind an orchestration instance directly by using the `func durable rewind` command in Core Tools.

NOTE

The Core Tools commands are currently only supported when using the default [Azure Storage provider](#) for persisting runtime state.

The `durable rewind` command takes the following parameters:

- `id` (**required**): ID of the orchestration instance.
- `reason` (**optional**): Reason for rewinding the orchestration instance.
- `connection-string-setting` (**optional**): Name of the application setting containing the storage connection string to use. The default is `AzureWebJobsStorage`.
- `task-hub-name` (**optional**): Name of the Durable Functions task hub to use. By default, the task hub name in the `host.json` file is used.

```
func durable rewind --id 0ab8c55a66644d68a3a8b220b12d209c --reason "Orchestrator failed and needs to be revived."
```

Purge instance history

To remove all the data associated with an orchestration, you can purge the instance history. For example, you might want to delete any storage resources associated with a completed instance. To do so, use the *purge instance* API defined by the [orchestration client](#).

This first example shows how to purge a single orchestration instance.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

```
[FunctionName("PurgeInstanceHistory")]
public static Task Run(
    [DurableClient] IDurableOrchestrationClient client,
    [QueueTrigger("purge-queue")] string instanceId)
{
    return client.PurgeInstanceHistoryAsync(instanceId);
}
```

The next example shows a timer-triggered function that purges the history for all orchestration instances that completed after the specified time interval. In this case, it removes data for all instances completed 30 or more days ago. This example function is scheduled to run once per day, at 12:00 PM UTC:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

```
[FunctionName("PurgeInstanceHistory")]
public static Task Run(
    [DurableClient] IDurableOrchestrationClient client,
    [TimerTrigger("0 0 12 * * *")] TimerInfo myTimer)
{
    return client.PurgeInstanceHistoryAsync(
        DateTime.MinValue,
        DateTime.UtcNow.AddDays(-30),
        new List<OrchestrationStatus>
        {
            OrchestrationStatus.Completed
        });
}
```

NOTE

The previous C# code is for Durable Functions 2.x. For Durable Functions 1.x, you must use `OrchestrationClient` attribute instead of the `DurableClient` attribute, and you must use the `DurableOrchestrationClient` parameter type instead of `IDurableOrchestrationClient`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

NOTE

For the purge history operation to succeed, the runtime status of the target instance must be `Completed`, `Terminated`, or `Failed`.

Azure Functions Core Tools

You can purge an orchestration instance's history by using the `func durable purge-history` command in Core Tools. Similar to the second C# example in the preceding section, it purges the history for all orchestration instances created during a specified time interval. You can further filter purged instances by runtime status.

NOTE

The Core Tools commands are currently only supported when using the default [Azure Storage provider](#) for persisting runtime state.

The `durable purge-history` command has several parameters:

- `created-after` **(optional)**: Purge the history of instances created after this date/time (UTC). ISO 8601 formatted datetimes accepted.
- `created-before` **(optional)**: Purge the history of instances created before this date/time (UTC). ISO 8601 formatted datetimes accepted.
- `runtime-status` **(optional)**: Purge the history of instances with a particular status (for example, running or completed). Can provide multiple (space separated) statuses.
- `connection-string-setting` **(optional)**: Name of the application setting containing the storage connection string to use. The default is `AzureWebJobsStorage`.
- `task-hub-name` **(optional)**: Name of the Durable Functions task hub to use. By default, the task hub name in the `host.json` file is used.

The following command deletes the history of all failed instances created before November 14, 2021 at 7:35 PM (UTC).

```
func durable purge-history --created-before 2021-11-14T19:35:00.000000Z --runtime-status failed
```

Delete a task hub

Using the [func durable delete-task-hub](#) command in Core Tools, you can delete all storage artifacts associated with a particular task hub, including Azure storage tables, queues, and blobs.

NOTE

The Core Tools commands are currently only supported when using the default [Azure Storage provider](#) for persisting runtime state.

The [durable delete-task-hub](#) command has two parameters:

- `connection-string-setting` **(optional)**: Name of the application setting containing the storage connection string to use. The default is `AzureWebJobsStorage`.
- `task-hub-name` **(optional)**: Name of the Durable Functions task hub to use. By default, the task hub name in the `host.json` file is used.

The following command deletes all Azure storage data associated with the `UserTest` task hub.

```
func durable delete-task-hub --task-hub-name UserTest
```

Next steps

[Learn how to handle versioning](#)

[Built-in HTTP API reference for instance management](#)

Versioning in Durable Functions (Azure Functions)

10/5/2022 • 6 minutes to read • [Edit Online](#)

It is inevitable that functions will be added, removed, and changed over the lifetime of an application. [Durable Functions](#) allows chaining functions together in ways that weren't previously possible, and this chaining affects how you can handle versioning.

How to handle breaking changes

There are several examples of breaking changes to be aware of. This article discusses the most common ones. The main theme behind all of them is that both new and existing function orchestrations are impacted by changes to function code.

Changing activity or entity function signatures

A signature change refers to a change in the name, input, or output of a function. If this kind of change is made to an activity or entity function, it could break any orchestrator function that depends on it. This is especially true for type-safe languages. If you update the orchestrator function to accommodate this change, you could break existing in-flight instances.

As an example, suppose we have the following orchestrator function.

- [C#](#)
- [Java](#)

```
[FunctionName("FooBar")]
public static Task Run([OrchestrationTrigger] IDurableOrchestrationContext context)
{
    bool result = await context.CallActivityAsync<bool>("Foo");
    await context.CallActivityAsync("Bar", result);
}
```

This simplistic function takes the results of **Foo** and passes it to **Bar**. Let's assume we need to change the return value of **Foo** from a Boolean to a String to support a wider variety of result values. The result looks like this:

- [C#](#)
- [Java](#)

```
[FunctionName("FooBar")]
public static Task Run([OrchestrationTrigger] IDurableOrchestrationContext context)
{
    string result = await context.CallActivityAsync<string>("Foo");
    await context.CallActivityAsync("Bar", result);
}
```

This change works fine for all new instances of the orchestrator function but breaks any in-flight instances. For example, consider the case where an orchestration instance calls a function named **Foo**, gets back a boolean value, and then checkpoints. If the signature change is deployed at this point, the checkpointed instance will fail immediately when it resumes and replays the call to **Foo**. This failure happens because the result in the history table is a Boolean value but the new code tries to deserialize it into a String value, resulting in a runtime exception for type-safe languages.

This example is just one of many different ways that a function signature change can break existing instances. In general, if an orchestrator needs to change the way it calls a function, then the change is likely to be problematic.

Changing orchestrator logic

The other class of versioning problems come from changing the orchestrator function code in a way that changes the execution path for in-flight instances.

Consider the following orchestrator function:

- [C#](#)
- [Java](#)

```
[FunctionName("FooBar")]
public static Task Run([OrchestrationTrigger] IDurableOrchestrationContext context)
{
    bool result = await context.CallActivityAsync<bool>("Foo");
    await context.CallActivityAsync("Bar", result);
}
```

Now let's assume you want to make a change to add a new function call in between the two existing function calls.

- [C#](#)
- [Java](#)

```
[FunctionName("FooBar")]
public static Task Run([OrchestrationTrigger] IDurableOrchestrationContext context)
{
    bool result = await context.CallActivityAsync<bool>("Foo");
    if (result)
    {
        await context.CallActivityAsync("SendNotification");
    }

    await context.CallActivityAsync("Bar", result);
}
```

This change adds a new function call to *SendNotification* between *Foo* and *Bar*. There are no signature changes. The problem arises when an existing instance resumes from the call to *Bar*. During replay, if the original call to *Foo* returned `true`, then the orchestrator replay will call into *SendNotification*, which is not in its execution history. The runtime detects this inconsistency and raises a *non-deterministic orchestration* error because it encountered a call to *SendNotification* when it expected to see a call to *Bar*. The same type of problem can occur when adding API calls to other durable operations, like creating durable timers, waiting for external events, calling sub-orchestrations, etc.

Mitigation strategies

Here are some of the strategies for dealing with versioning challenges:

- Do nothing (not recommended)
- Stop all in-flight instances
- Side-by-side deployments

Do nothing

The naive approach to versioning is to do nothing and let in-flight orchestration instances fail. Depending on the

type of change, the following types of failures may occur.

- Orchestrations may fail with a *non-deterministic orchestration* error.
- Orchestrations may get stuck indefinitely, reporting a `Running` status.
- If a function gets removed, any function that tries to call it may fail with an error.
- If a function gets removed after it was scheduled to run, then the app may experience low-level runtime failures in the Durable Task Framework engine, potentially resulting in severe performance degradation.

Because of these potential failures, the "do nothing" strategy is not recommended.

Stop all in-flight instances

Another option is to stop all in-flight instances. If you're using the default [Azure Storage provider for Durable Functions](#), stopping all instances can be done by clearing the contents of the internal `control-queue` and `workitem-queue` queues. You can alternatively stop the function app, delete these queues, and restart the app again. The queues will be recreated automatically once the app restarts. The previous orchestration instances may remain in the "Running" state indefinitely, but they will not clutter your logs with failure messages or cause any harm to your app. This approach is ideal in rapid prototype development, including local development.

NOTE

This approach requires direct access to the underlying storage resources, and may not be appropriate for all storage providers supported by Durable Functions.

Side-by-side deployments

The most fail-proof way to ensure that breaking changes are deployed safely is by deploying them side-by-side with your older versions. This can be done using any of the following techniques:

- Deploy all the updates as entirely new functions, leaving existing functions as-is. This generally isn't recommended because of the complexity involved in recursively updating the callers of the new function versions.
- Deploy all the updates as a new function app with a different storage account.
- Deploy a new copy of the function app with the same storage account but with an updated [task hub](#) name. This results in the creation of new storage artifacts that can be used by the new version of your app. The old version of your app will continue to execute using the previous set of storage artifacts.

Side-by-side deployment is the recommended technique for deploying new versions of your function apps.

NOTE

This guidance for the side-by-side deployment strategy uses Azure Storage-specific terms, but applies generally to all supported [Durable Functions storage providers](#).

Deployment slots

When doing side-by-side deployments in Azure Functions or Azure App Service, we recommend that you deploy the new version of the function app to a new [Deployment slot](#). Deployment slots allow you to run multiple copies of your function app side-by-side with only one of them as the active *production* slot. When you are ready to expose the new orchestration logic to your existing infrastructure, it can be as simple as swapping the new version into the production slot.

NOTE

This strategy works best when you use HTTP and webhook triggers for orchestrator functions. For non-HTTP triggers, such as queues or Event Hubs, the trigger definition should [derive from an app setting](#) that gets updated as part of the swap operation.

Next steps

[Learn about using and choosing storage providers](#)

Durable Functions storage providers

10/5/2022 • 9 minutes to read • [Edit Online](#)

Durable Functions is a set of Azure Functions triggers and bindings that are internally powered by the [Durable Task Framework](#) (DTFx). DTFx supports various backend storage providers, including the Azure Storage provider used by Durable Functions. Starting in Durable Functions [v2.5.0](#), users can configure their function apps to use DTFx storage providers other than the Azure Storage provider.

NOTE

For many function apps, the default Azure Storage provider for Durable Functions is likely to suffice, and is the easiest to use since it requires no extra configuration. However, there are cost, scalability, and data management tradeoffs that may favor the use of an alternate storage provider.

Two alternate storage providers were developed for use with Durable Functions and the Durable Task Framework, namely the *Netherite* storage provider and the *Microsoft SQL Server (MSSQL)* storage provider. This article describes all three supported providers, compares them against each other, and provides basic information about how to get started using them.

NOTE

It's not currently possible to migrate data from one storage provider to another. If you want to use a new storage provider, you should create a new app configured with the new storage provider.

Azure Storage

Azure Storage is the default storage provider for Durable Functions. It uses queues, tables, and blobs to persist orchestration and entity state. It also uses blobs and blob leases to manage partitions. In many cases, the storage account used to store Durable Functions runtime state is the same as the default storage account used by Azure Functions (`AzureWebJobsStorage`). However, it's also possible to configure Durable Functions with a separate storage account. The Azure Storage provider is built-into the Durable Functions extension and doesn't have any other dependencies.

The key benefits of the Azure Storage provider include:

- No setup required - you can use the storage account that was created for you by the function app setup experience.
- Lowest-cost serverless billing model - Azure Storage has a consumption-based pricing model based entirely on usage ([more information](#)).
- Best tooling support - Azure Storage offers cross-platform local emulation and integrates with Visual Studio, Visual Studio Code, and the Azure Functions Core Tools.
- Most mature - Azure Storage was the original and most battle-tested storage backend for Durable Functions.
- Preview support for using identity instead of secrets for connecting to the storage provider.

The source code for the DTFx components of the Azure Storage storage provider can be found in the [Azure/durabletask](#) GitHub repo.

NOTE

Standard general purpose Azure Storage accounts are required when using the Azure Storage provider. All other storage account types are not supported. We highly recommend using legacy v1 general purpose storage accounts because the newer v2 storage accounts can be significantly more expensive for Durable Functions workloads. For more information on Azure Storage account types, see the [Storage account overview](#) documentation.

Netherite (preview)

The Netherite storage backend was designed and developed by [Microsoft Research](#). It uses [Azure Event Hubs](#) and the [FASTER](#) database technology on top of [Azure Page Blobs](#). The design of Netherite enables significantly higher-throughput processing of orchestrations and entities compared to other providers. In some benchmark scenarios, throughput was shown to increase by more than an order of magnitude when compared to the default Azure Storage provider.

The key benefits of the Netherite storage provider include:

- Significantly higher throughput at lower cost compared to other storage providers.
- Supports price-performance optimization, allowing you to scale-up performance as-needed.
- Supports up to 32 data partitions with Event Hubs Basic and Standard SKUs.
- More cost-effective than other providers for high-throughput workloads.

You can learn more about the technical details of the Netherite storage provider, including how to get started using it, in the [Netherite documentation](#). The source code for the Netherite storage provider can be found in the [microsoft/durabletask-netherite](#) GitHub repo. A more in-depth evaluation of the Netherite storage provider is also available in the following research paper: [Serverless Workflows with Durable Functions and Netherite](#).

NOTE

The *Netherite* name originates from the world of [Minecraft](#).

Microsoft SQL Server (MSSQL) (preview)

The Microsoft SQL Server (MSSQL) storage provider persists all state into a Microsoft SQL Server database. It's compatible with both on-premises and cloud-hosted deployments of SQL Server, including [Azure SQL Database](#).

The key benefits of the MSSQL storage provider include:

- Supports disconnected environments - no Azure connectivity is required when using a SQL Server installation.
- Portable across multiple environments and clouds, including Azure-hosted and on-premises.
- Strong data consistency, enabling backup/restore and failover without data loss.
- Native support for custom data encryption (a feature of SQL Server).
- Integrates with existing database applications via built-in stored procedures.

You can learn more about the technical details of the MSSQL storage provider, including how to get started using it, in the [Microsoft SQL provider documentation](#). The source code for the MSSQL storage provider can be found in the [microsoft/durabletask-mssql](#) GitHub repo.

Configuring alternate storage providers

Configuring alternate storage providers is generally a two-step process:

1. Add the appropriate NuGet package to your function app (this requirement is temporary for apps using extension bundles).

2. Update the `host.json` file to specify which storage provider you want to use.

If no storage provider is explicitly configured in `host.json`, the Azure Storage provider will be enabled by default.

Configuring the Azure storage provider

The Azure Storage provider is the default storage provider and doesn't require any explicit configuration, NuGet package references, or extension bundle references. You can find the full set of `host.json` configuration options [here](#), under the `extensions/durableTask/storageProvider` path.

Connections

The `connectionName` property in `host.json` is a reference to environment configuration which specifies how the app should connect to Azure Storage. It may specify:

- The name of an application setting containing a connection string. To obtain a connection string, follow the steps shown at [Manage storage account access keys](#).
- The name of a shared prefix for multiple application settings, together defining an [identity-based connection](#).

If the configured value is both an exact match for a single setting and a prefix match for other settings, the exact match is used. If no value is specified in `host.json`, the default value is "AzureWebJobsStorage".

Identity-based connections (preview)

If you are using [version 2.7.0 or higher of the extension](#) and the Azure storage provider, instead of using a connection string with a secret, you can have the app use an [Azure Active Directory identity](#). To do this, you would define settings under a common prefix which maps to the `connectionName` property in the trigger and binding configuration.

To use an identity-based connection for Durable Functions, configure the following app settings:

PROPERTY	ENVIRONMENT VARIABLE TEMPLATE	DESCRIPTION	EXAMPLE VALUE
Blob service URI	<code><CONNECTION_NAME_PREFIX>__blob</code>	The <code>data plane</code> URI of the blob service of the storage account, using the HTTPS scheme.	<code>https://<storage_account_name>.blob.core.windows.net</code>
Queue service URI	<code><CONNECTION_NAME_PREFIX>__queue</code>	The <code>data plane</code> URI of the queue service of the storage account, using the HTTPS scheme.	<code>https://<storage_account_name>.queue.core.windows.net</code>
Table service URI	<code><CONNECTION_NAME_PREFIX>__table</code>	The <code>data plane</code> URI of a table service of the storage account, using the HTTPS scheme.	<code>https://<storage_account_name>.table.core.windows.net</code>

Additional properties may be set to customize the connection. See [Common properties for identity-based connections](#).

When hosted in the Azure Functions service, identity-based connections use a [managed identity](#). The system-assigned identity is used by default, although a user-assigned identity can be specified with the `credential` and `clientID` properties. Note that configuring a user-assigned identity with a resource ID is **not** supported. When run in other contexts, such as local development, your developer identity is used instead, although this can be customized. See [Local development with identity-based connections](#).

Grant permission to the identity

Whatever identity is being used must have permissions to perform the intended actions. You will need to [assign a role in Azure RBAC](#), using either built-in or custom roles which provide those permissions.

IMPORTANT

Some permissions might be exposed by the target service that are not necessary for all contexts. Where possible, adhere to the **principle of least privilege**, granting the identity only required privileges. For example, if the app only needs to be able to read from a data source, use a role that only has permission to read. It would be inappropriate to assign a role that also allows writing to that service, as this would be excessive permission for a read operation. Similarly, you would want to ensure the role assignment is scoped only over the resources that need to be read.

You'll need to create a role assignment that provides access to Azure storage at runtime. Management roles like [Owner](#) aren't sufficient. The following built-in roles are recommended when using the Durable Functions extension in normal operation:

- [Storage Blob Data Contributor](#)
- [Storage Queue Data Contributor](#)
- [Storage Table Data Contributor](#)

Your application may require more permissions based on the code you write. If you're using the default behavior or explicitly setting `connectionName` to "AzureWebJobsStorage", see [Connecting to host storage with an identity](#) for other permission considerations.

Configuring the Netherite storage provider

To use the Netherite storage provider, you must first add a reference to the [Microsoft.Azure.DurableTask.Netherite.AzureFunctions](#) NuGet package in your `csproj` file (.NET apps) or your `extensions.proj` file (JavaScript, Python, and PowerShell apps).

The following `host.json` example shows the minimum configuration required to enable the Netherite storage provider.

```
{
  "version": "2.0",
  "extensions": {
    "durableTask": {
      "storageProvider": {
        "type": "Netherite",
        "storageConnectionName": "AzureWebJobsStorage",
        "eventHubsConnectionName": "EventHubsConnection"
      }
    }
  }
}
```

For more detailed setup instructions, see the [Netherite getting started documentation](#).

Configuring the MSSQL storage provider

To use the MSSQL storage provider, you must first add a reference to the [Microsoft.DurableTask.SqlServer.AzureFunctions](#) NuGet package in your `csproj` file (.NET apps) or your `extensions.proj` file (JavaScript, Python, and PowerShell apps).

NOTE

The MSSQL storage provider is not yet supported in apps that use [extension bundles](#).

The following example shows the minimum configuration required to enable the MSSQL storage provider.

```
{
  "version": "2.0",
  "extensions": {
    "durableTask": {
      "storageProvider": {
        "type": "mssql",
        "connectionStringName": "SQLDB_Connection"
      }
    }
  }
}
```

For more detailed setup instructions, see the [MSSQL provider's getting started documentation](#).

Comparing storage providers

There are many significant tradeoffs between the various supported storage providers. The following table can be used to help you understand these tradeoffs and decide which storage provider is best for your needs.

STORAGE PROVIDER	AZURE STORAGE	NETHERITE	MSSQL
Official support status	<input checked="" type="checkbox"/> Generally available (GA)	<input type="triangle-down"/> Public preview	<input type="triangle-down"/> Public preview
External dependencies	Azure Storage account (general purpose v1)	Azure Event Hubs Azure Storage account (general purpose)	SQL Server 2019 or Azure SQL Database
Local development and emulation options	Azurite v3.12+ (cross platform) Azure Storage Emulator (Windows only)	Supports in-memory emulation of task hubs (more information)	SQL Server Developer Edition (supports Windows , Linux , and Docker containers)
Task hub configuration	Explicit	Explicit	Implicit by default (more information)
Maximum throughput	Moderate	Very high	Moderate
Maximum orchestration/entity scale-out (nodes)	16	32	N/A
Maximum activity scale-out (nodes)	N/A	32	N/A
Consumption plan support	<input checked="" type="checkbox"/> Fully supported	<input type="times"/> Not supported	<input type="times"/> Not supported
Elastic Premium plan support	<input checked="" type="checkbox"/> Fully supported	<input type="triangle-down"/> Requires runtime scale monitoring	<input type="triangle-down"/> Requires runtime scale monitoring
KEDA 2.0 scaling support (more information)	<input type="times"/> Not supported	<input type="times"/> Not supported	<input checked="" type="checkbox"/> Supported using the MSSQL scaler (more information)
Support for extension bundles (recommended for non-.NET apps)	<input checked="" type="checkbox"/> Fully supported	<input type="times"/> Not supported	<input type="times"/> Not supported

STORAGE PROVIDER	AZURE STORAGE	NETHERITE	MSSQL
Price-performance configurable?	✗ No	<input checked="" type="checkbox"/> Yes (Event Hubs TUs and CUs)	<input checked="" type="checkbox"/> Yes (SQL vCPUs)
Managed Identity Support	<input checked="" type="checkbox"/> Fully supported	✗ Not supported	⚠ Requires runtime-driven scaling
Disconnected environment support	✗ Azure connectivity required	✗ Azure connectivity required	<input checked="" type="checkbox"/> Fully supported
Identity-based connections	<input checked="" type="checkbox"/> Yes (preview)	✗ No	✗ No

Next steps

[Learn about Durable Functions performance and scale](#)

Performance and scale in Durable Functions (Azure Functions)

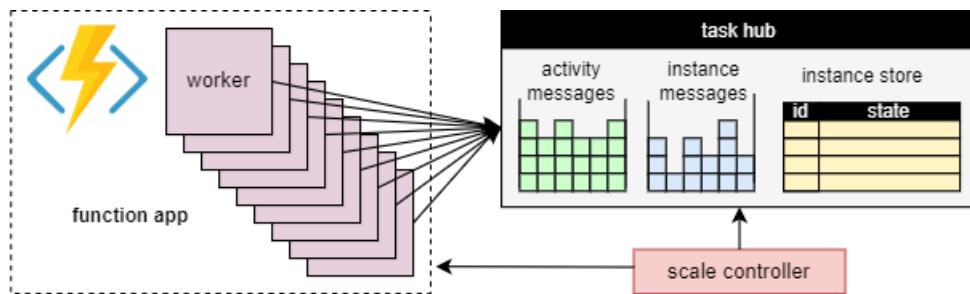
10/5/2022 • 10 minutes to read • [Edit Online](#)

To optimize performance and scalability, it's important to understand the unique scaling characteristics of [Durable Functions](#). In this article, we explain how workers are scaled based on load, and how one can tune the various parameters.

Worker scaling

A fundamental benefit of the [task hub concept](#) is that the number of workers that process task hub work items can be continuously adjusted. In particular, applications can add more workers (*scale out*) if the work needs to be processed more quickly, and can remove workers (*scale in*) if there is not enough work to keep the workers busy. It is even possible to *scale to zero* if the task hub is completely idle. When scaled to zero, there are no workers at all; only the scale controller and the storage need to remain active.

The following diagram illustrates this concept:



Automatic scaling

As with all Azure Functions running in the Consumption and Elastic Premium plans, Durable Functions supports auto-scale via the [Azure Functions scale controller](#). The Scale Controller monitors how long messages and tasks have to wait before they are processed. Based on these latencies it can decide whether to add or remove workers.

NOTE

Starting with Durable Functions 2.0, function apps can be configured to run within VNET-protected service endpoints in the Elastic Premium plan. In this configuration, the Durable Functions triggers initiate scale requests instead of the Scale Controller. For more information, see [Runtime scale monitoring](#).

On a premium plan, automatic scaling can help to keep the number of workers (and therefore the operating cost) roughly proportional to the load that the application is experiencing.

CPU usage

Orchestrator functions are executed on a single thread to ensure that execution can be deterministic across many replays. Because of this single-threaded execution, it's important that orchestrator function threads do not perform CPU-intensive tasks, do I/O, or block for any reason. Any work that may require I/O, blocking, or multiple threads should be moved into activity functions.

Activity functions have all the same behaviors as regular queue-triggered functions. They can safely do I/O, execute CPU intensive operations, and use multiple threads. Because activity triggers are stateless, they can

freely scale out to an unbounded number of VMs.

Entity functions are also executed on a single thread and operations are processed one-at-a-time. However, entity functions do not have any restrictions on the type of code that can be executed.

Function timeouts

Activity, orchestrator, and entity functions are subject to the same [function timeouts](#) as all Azure Functions. As a general rule, Durable Functions treats function timeouts the same way as unhandled exceptions thrown by the application code.

For example, if an activity times out, the function execution is recorded as a failure, and the orchestrator is notified and handles the timeout just like any other exception: retries take place if specified by the call, or an exception handler may be executed.

Entity operation batching

To improve performance and reduce cost, a single work item may execute an entire batch of entity operations. On consumption plans, each batch is then billed as a single function execution.

By default, the maximum batch size is 50 for consumption plans and 5000 for all other plans. The maximum batch size can also be configured in the [host.json](#) file. If the maximum batch size is 1, batching is effectively disabled.

NOTE

If individual entity operations take a long time to execute, it may be beneficial to limit the maximum batch size to reduce the risk of [function timeouts](#), in particular on consumption plans.

Instance caching

Generally, to process an [orchestration work item](#), a worker has to both

1. Fetch the orchestration history.
2. Replay the orchestrator code using the history.

If the same worker is processing multiple work items for the same orchestration, the storage provider can optimize this process by caching the history in the worker's memory, which eliminates the first step. Moreover, it can cache the mid-execution orchestrator, which eliminates the second step, the history replay, as well.

The typical effect of caching is reduced I/O against the underlying storage service, and overall improved throughput and latency. On the other hand, caching increases the memory consumption on the worker.

Instance caching is currently supported by the Azure Storage provider and by the Netherite storage provider. The table below provides a comparison.

	AZURE STORAGE PROVIDER	NETHERITE STORAGE PROVIDER	MSSQL STORAGE PROVIDER
Instance caching	Supported (.NET in-process worker only)	Supported	Not supported
Default setting	Disabled	Enabled	n/a
Mechanism	Extended Sessions	Instance Cache	n/a
Documentation	See Extended sessions	See Instance cache	n/a

TIP

Caching can reduce how often histories are replayed, but it cannot eliminate replay altogether. When developing orchestrators, we highly recommend testing them on a configuration that disables caching. This forced-replay behavior can be useful for detecting [orchestrator function code constraints violations](#) at development time.

Comparison of caching mechanisms

The providers use different mechanisms to implement caching, and offer different parameters to configure the caching behavior.

- **Extended sessions**, as used by the Azure Storage provider, keep mid-execution orchestrators in memory until they are idle for some time. The parameters to control this mechanism are `extendedSessionsEnabled` and `extendedSessionIdleTimeoutInSeconds`. For more details, see the section [Extended sessions](#) of the Azure Storage provider documentation.

NOTE

Extended sessions are supported only in the .NET in-process worker.

- The **Instance cache**, as used by the Netherite storage provider, keeps the state of all instances, including their histories, in the worker's memory, while keeping track of the total memory used. If the cache size exceeds the limit configured by `InstanceCacheSizeMB`, the least recently used instance data is evicted. If `CacheOrchestrationCursors` is set to true, the cache also stores the mid-execution orchestrators along with the instance state. For more details, see the section [Instance cache](#) of the Netherite storage provider documentation.

NOTE

Instance caches work for all language SDKs, but the `CacheOrchestrationCursors` option is available only for the .NET in-process worker.

Concurrency throttles

A single worker instance can execute multiple [work items](#) concurrently. This helps to increase parallelism and more efficiently utilize the workers. However, if a worker attempts to process too many work items at the same time, it may exhaust its available resources, such as the CPU load, the number of network connections, or the available memory.

To ensure that an individual worker does not overcommit, it may be necessary to throttle the per-instance concurrency. By limiting the number of functions that are concurrently running on each worker, we can avoid exhausting the resource limits on that worker.

NOTE

The concurrency throttles only apply locally, to limit what is currently being processed **per worker**. Thus, these throttles do not limit the total throughput of the system.

TIP

In some cases, throttling the per-worker concurrency can actually *increase* the total throughput of the system. This can occur when each worker takes less work, causing the scale controller to add more workers to keep up with the queues, which then increases the total throughput.

Configuration of throttles

Activity, orchestrator, and entity function concurrency limits can be configured in the `host.json` file. The relevant settings are `durableTask/maxConcurrentActivityFunctions` for activity functions and `durableTask/maxConcurrentOrchestratorFunctions` for both orchestrator and entity functions. These settings control the maximum number of orchestrator, entity, or activity functions that are loaded into memory on a single worker.

NOTE

Orchestrations and entities are only loaded into memory when they are actively processing events or operations, or if [instance caching](#) is enabled. After executing their logic and awaiting (i.e. hitting an `await` (C#) or `yield` (JavaScript, Python) statement in the orchestrator function code), they may be unloaded from memory. Orchestrations and entities that are unloaded from memory don't count towards the `maxConcurrentOrchestratorFunctions` throttle. Even if millions of orchestrations or entities are in the "Running" state, they only count towards the throttle limit when they are loaded into active memory. An orchestration that schedules an activity function similarly doesn't count towards the throttle if the orchestration is waiting for the activity to finish executing.

Functions 2.0

```
{  
  "extensions": {  
    "durableTask": {  
      "maxConcurrentActivityFunctions": 10,  
      "maxConcurrentOrchestratorFunctions": 10  
    }  
  }  
}
```

Functions 1.x

```
{  
  "durableTask": {  
    "maxConcurrentActivityFunctions": 10,  
    "maxConcurrentOrchestratorFunctions": 10  
  }  
}
```

Language runtime considerations

The language runtime you select may impose strict concurrency restrictions on your functions. For example, Durable Function apps written in Python or PowerShell may only support running a single function at a time on a single VM. This can result in significant performance problems if not carefully accounted for. For example, if an orchestrator fans-out to 10 activities but the language runtime restricts concurrency to just one function, then 9 of the 10 activity functions will be stuck waiting for a chance to run. Furthermore, these 9 stuck activities will not be able to be load balanced to any other workers because the Durable Functions runtime will have already loaded them into memory. This becomes especially problematic if the activity functions are long-running.

If the language runtime you are using places a restriction on concurrency, you should update the Durable Functions concurrency settings to match the concurrency settings of your language runtime. This ensures that the Durable Functions runtime will not attempt to run more functions concurrently than is allowed by the

language runtime, allowing any pending activities to be load balanced to other VMs. For example, if you have a Python app that restricts concurrency to 4 functions (perhaps it's only configured with 4 threads on a single language worker process or 1 thread on 4 language worker processes) then you should configure both `maxConcurrentOrchestratorFunctions` and `maxConcurrentActivityFunctions` to 4.

For more information and performance recommendations for Python, see [Improve throughput performance of Python apps in Azure Functions](#). The techniques mentioned in this Python developer reference documentation can have a substantial impact on Durable Functions performance and scalability.

Partition count

Some of the storage providers use a *partitioning* mechanism and allow specifying a `partitionCount` parameter.

When using partitioning, workers do not directly compete for individual work items. Instead, the work items are first grouped into `partitionCount` partitions. These partitions are then assigned to workers. This partitioned approach to load distribution can help to reduce the total number of storage accesses required. Also, it can enable [instance caching](#) and improve locality because it creates *affinity*: all work items for the same instance are processed by the same worker.

NOTE

Partitioning limits scale out because at most `partitionCount` workers can process work items from a partitioned queue.

The following table shows, for each storage provider, which queues are partitioned, and the allowable range and default values for the `partitionCount` parameter.

	AZURE STORAGE PROVIDER	NETHERITE STORAGE PROVIDER	MSSQL STORAGE PROVIDER
Instance messages	Partitioned	Partitioned	Not partitioned
Activity messages	Not partitioned	Partitioned	Not partitioned
Default <code>partitionCount</code>	4	12	n/a
Maximum <code>partitionCount</code>	16	32	n/a
Documentation	See Orchestrator scale-out	See Partition count considerations	n/a

WARNING

The partition count can no longer be changed after a task hub has been created. Thus, it is advisable to set it to a large enough value to accommodate future scale out requirements for the task hub instance.

Configuration of partition count

The `partitionCount` parameter can be specified in the `host.json` file. The following example host.json snippet sets the `durableTask/storageProvider/partitionCount` property (or `durableTask/partitionCount` in Durable Functions 1.x) to 3.

Durable Functions 2.x

```
{
  "extensions": {
    "durableTask": {
      "storageProvider": {
        "partitionCount": 3
      }
    }
  }
}
```

Durable Functions 1.x

```
{
  "extensions": {
    "durableTask": {
      "partitionCount": 3
    }
  }
}
```

Performance targets

When planning to use Durable Functions for a production application, it is important to consider the performance requirements early in the planning process. Some basic usage scenarios include:

- **Sequential activity execution:** This scenario describes an orchestrator function that runs a series of activity functions one after the other. It most closely resembles the [Function Chaining](#) sample.
- **Parallel activity execution:** This scenario describes an orchestrator function that executes many activity functions in parallel using the [Fan-out, Fan-in](#) pattern.
- **Parallel response processing:** This scenario is the second half of the [Fan-out, Fan-in](#) pattern. It focuses on the performance of the fan-in. It's important to note that unlike fan-out, fan-in is done by a single orchestrator function instance, and therefore can only run on a single VM.
- **External event processing:** This scenario represents a single orchestrator function instance that waits on [external events](#), one at a time.
- **Entity operation processing:** This scenario tests how quickly a *single* [Counter entity](#) can process a constant stream of operations.

We provide throughput numbers for these scenarios in the respective documentation for the storage providers. In particular:

- for the Azure Storage provider, see [Performance Targets](#).
- for the Netherite storage provider, see [Basic Scenarios](#).
- for the MSSQL storage provider, see [Orchestration Throughput Benchmarks](#).

TIP

Unlike fan-out, fan-in operations are limited to a single VM. If your application uses the fan-out, fan-in pattern and you are concerned about fan-in performance, consider sub-dividing the activity function fan-out across multiple [sub-orchestrations](#).

Next steps

[Learn about the Azure Storage provider](#)

Disaster recovery and geo-distribution in Azure Durable Functions

10/5/2022 • 7 minutes to read • [Edit Online](#)

Microsoft strives to ensure that Azure services are always available. However, unplanned service outages may occur. If your application requires resiliency, Microsoft recommends configuring your app for geo-redundancy. Additionally, customers should have a disaster recovery plan in place for handling a regional service outage. An important part of a disaster recovery plan is preparing to fail over to the secondary replica of your app and storage in the event that the primary replica becomes unavailable.

In Durable Functions, all state is persisted in Azure Storage by default. A [task hub](#) is a logical container for Azure Storage resources that are used for [orchestrations](#) and [entities](#). Orchestrator, activity, and entity functions can only interact with each other when they belong to the same task hub. This document will refer to task hubs when describing scenarios for keeping these Azure Storage resources highly available.

NOTE

The guidance in this article assumes that you are using the default Azure Storage provider for storing Durable Functions runtime state. However, it's possible to configure alternate storage providers that store state elsewhere, like a SQL Server database. Different disaster recovery and geo-distribution strategies may be required for the alternate storage providers. For more information on the alternate storage providers, see the [Durable Functions storage providers](#) documentation.

Orchestrations and entities can be triggered using [client functions](#) that are themselves triggered via HTTP or one of the other supported Azure Functions trigger types. They can also be triggered using [built-in HTTP APIs](#). For simplicity, this article will focus on scenarios involving Azure Storage and HTTP-based function triggers, and options to increase availability and minimize downtime during disaster recovery activities. Other trigger types, such as Service Bus or Cosmos DB triggers, will not be explicitly covered.

The following scenarios are based on Active-Passive configurations, since they are guided by the usage of Azure Storage. This pattern consists of deploying a backup (passive) function app to a different region. Traffic Manager will monitor the primary (active) function app for HTTP availability. It will fail over to the backup function app if the primary fails. For more information, see [Azure Traffic Manager's Priority Traffic-Routing Method](#).

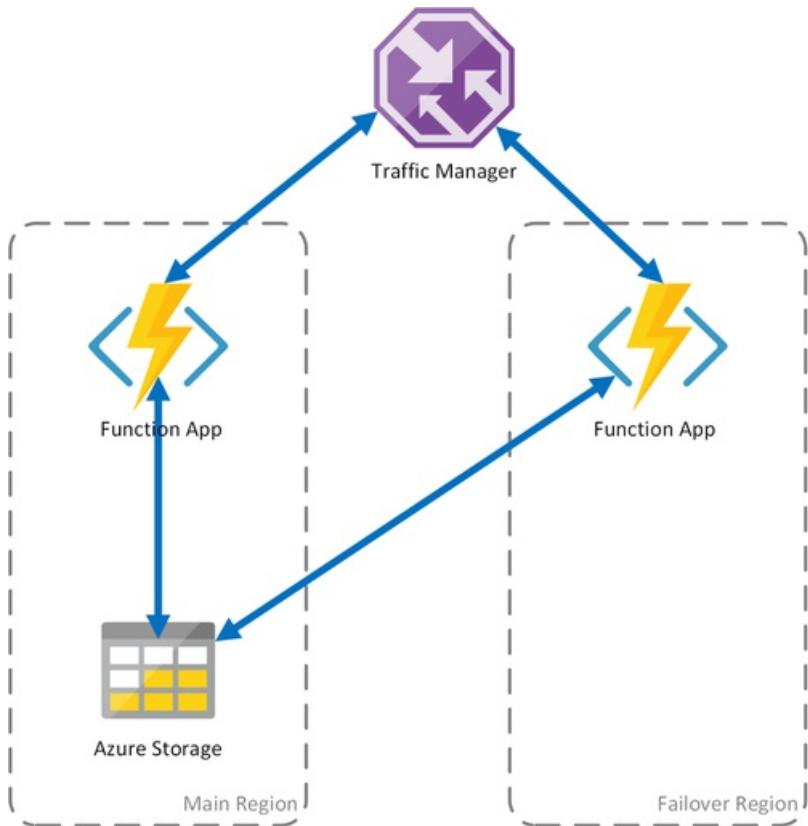
NOTE

- The proposed Active-Passive configuration ensures that a client is always able to trigger new orchestrations via HTTP. However, as a consequence of having two function apps sharing the same task hub in storage, some background storage transactions will be distributed between both of them. This configuration therefore incurs some added egress costs for the secondary function app.
- The underlying storage account and task hub are created in the primary region, and are shared by both function apps.
- All function apps that are redundantly deployed must share the same function access keys in the case of being activated via HTTP. The Functions Runtime exposes a [management API](#) that enables consumers to programmatically add, delete, and update function keys. Key management is also possible using [Azure Resource Manager APIs](#).

Scenario 1 - Load balanced compute with shared storage

If the compute infrastructure in Azure fails, the function app may become unavailable. To minimize the possibility of such downtime, this scenario uses two function apps deployed to different regions. Traffic Manager is configured to detect problems in the primary function app and automatically redirect traffic to the function

app in the secondary region. This function app shares the same Azure Storage account and Task Hub. Therefore, the state of the function apps isn't lost and work can resume normally. Once health is restored to the primary region, Azure Traffic Manager will start routing requests to that function app automatically.



There are several benefits when using this deployment scenario:

- If the compute infrastructure fails, work can resume in the failover region without data loss.
- Traffic Manager takes care of the automatic failover to the healthy function app automatically.
- Traffic Manager automatically re-establishes traffic to the primary function app after the outage has been corrected.

However, using this scenario consider:

- If the function app is deployed using a dedicated App Service plan, replicating the compute infrastructure in the failover datacenter increases costs.
- This scenario covers outages at the compute infrastructure, but the storage account continues to be the single point of failure for the function App. If a Storage outage occurs, the application suffers downtime.
- If the function app is failed over, there will be increased latency since it will access its storage account across regions.
- Accessing the storage service from a different region where it's located incurs in higher cost due to network egress traffic.
- This scenario depends on Traffic Manager. Considering [how Traffic Manager works](#), it may be some time until a client application that consumes a Durable Function needs to query again the function app address from Traffic Manager.

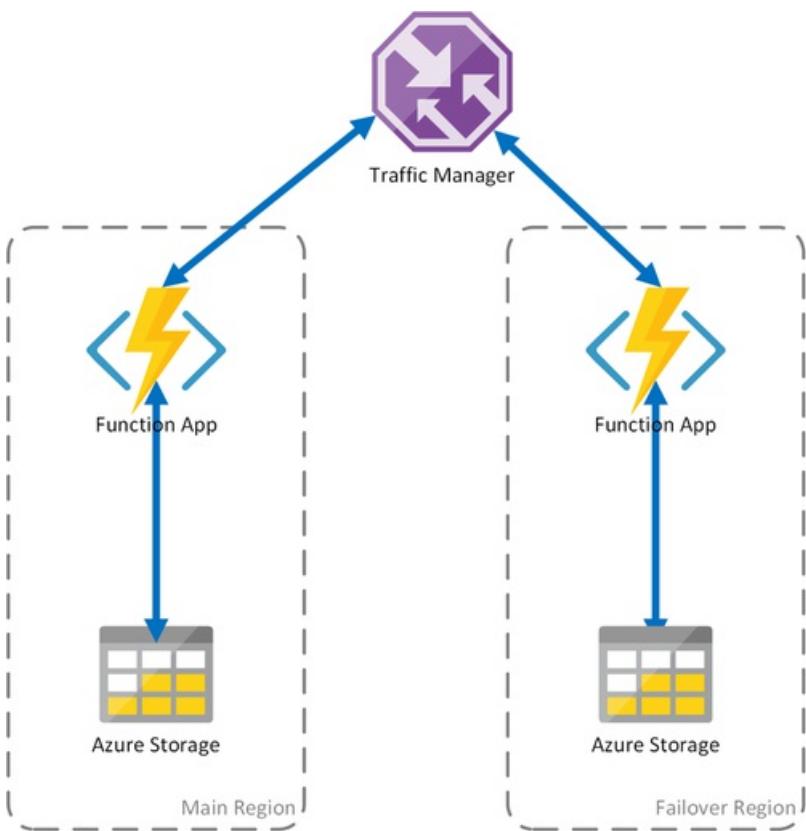
NOTE

Starting in v2.3.0 of the Durable Functions extension, two function apps can be run safely at the same time with the same storage account and task hub configuration. The first app to start will acquire an application-level blob lease that prevents other apps from stealing messages from the task hub queues. If this first app stops running, its lease will expire and can be acquired by a second app, which will then proceed to process task hub messages.

Prior to v2.3.0, function apps that are configured to use the same storage account will process messages and update storage artifacts concurrently, resulting in much higher overall latencies and egress costs. If the primary and replica apps ever have different code deployed to them, even temporarily, then orchestrations could also fail to execute correctly because of orchestrator function inconsistencies across the two apps. It is therefore recommended that all apps that require geo-distribution for disaster recovery purposes use v2.3.0 or higher of the Durable extension.

Scenario 2 - Load balanced compute with regional storage

The preceding scenario covers only the case of failure in the compute infrastructure. If the storage service fails, it will result in an outage of the function app. To ensure continuous operation of the durable functions, this scenario uses a local storage account on each region to which the function apps are deployed.



This approach adds improvements on the previous scenario:

- If the function app fails, Traffic Manager takes care of failing over to the secondary region. However, because the function app relies on its own storage account, the durable functions continue to work.
- During a failover, there is no additional latency in the failover region since the function app and the storage account are colocated.
- Failure of the storage layer will cause failures in the durable functions, which in turn will trigger a redirection to the failover region. Again, since the function app and storage are isolated per region, the durable functions will continue to work.

Important considerations for this scenario:

- If the function app is deployed using a dedicated App Service plan, replicating the compute infrastructure in

the failover datacenter increases costs.

- Current state isn't failed over, which implies that existing orchestrations and entities will be effectively paused and unavailable until the primary region recovers.

To summarize, the tradeoff between the first and second scenario is that latency is preserved and egress costs are minimized but existing orchestrations and entities will be unavailable during the downtime. Whether these tradeoffs are acceptable depends on the requirements of the application.

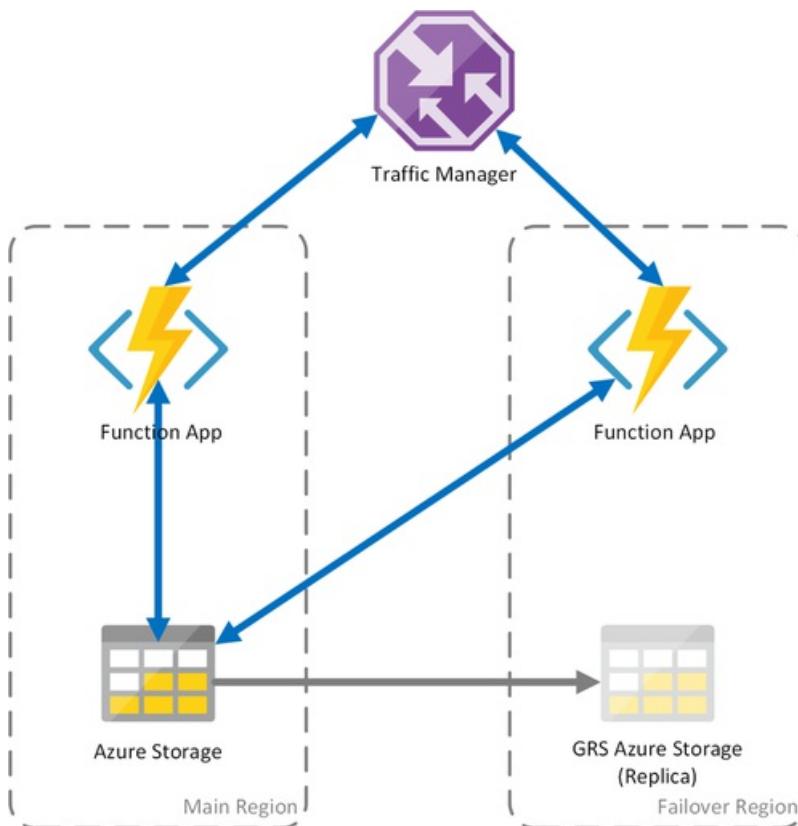
Scenario 3 - Load balanced compute with GRS shared storage

This scenario is a modification over the first scenario, implementing a shared storage account. The main difference is that the storage account is created with geo-replication enabled. Functionally, this scenario provides the same advantages as Scenario 1, but it enables additional data recovery advantages:

- Geo-redundant storage (GRS) and Read-access GRS (RA-GRS) maximize availability for your storage account.
- If there is a regional outage of the Storage service, you can [manually initiate a failover to the secondary replica](#). In extreme circumstances where a region is lost due to a significant disaster, Microsoft may initiate a regional failover. In this case, no action on your part is required.
- When a failover happens, state of the durable functions will be preserved up to the last replication of the storage account, which typically occurs every few minutes.

As with the other scenarios, there are important considerations:

- A failover to the replica may take some time. Until the failover completes and Azure Storage DNS records have been updated, the function app will suffer an outage.
- There is an increased cost for using geo-replicated storage accounts.
- GRS replication copies your data asynchronously. Some of the latest transactions might be lost because of the latency of the replication process.



NOTE

As described in Scenario 1, it is strongly recommended that function apps deployed with this strategy use v2.3.0 or higher of the Durable Functions extension.

For more information, see the [Azure Storage disaster recovery and storage account failover](#) documentation.

Next steps

[Learn more about designing highly available applications in Azure Storage](#)

Data persistence and serialization in Durable Functions (Azure Functions)

10/5/2022 • 5 minutes to read • [Edit Online](#)

The Durable Functions runtime automatically persists function parameters, return values, and other state to the [task hub](#) in order to provide reliable execution. However, the amount and frequency of data persisted to durable storage can impact application performance and storage transaction costs. Depending on the type of data your application stores, data retention and privacy policies may also need to be considered.

Task Hub Contents

Task hubs store the current state of instances, and any pending messages:

- *Instance states* store the current status and history of an instance. For orchestration instances, this includes the runtime state, the orchestration history, inputs, outputs, and custom status. For entity instances, it includes the entity state.
- *Messages* store function inputs or outputs, event payloads, and metadata that Durable Functions uses for internal purposes, like routing and end-to-end correlation.

Messages are deleted after being processed, but instance states persist unless they're explicitly deleted by the application or an operator. In particular, an orchestration history remains in storage even after the orchestration completes.

For an example of how states and messages represent the progress of an orchestration, see the [task hub execution example](#).

Where and how states and messages are represented in storage [depends on the storage provider](#). By default, Durable Functions uses the [Azure Storage provider](#) which persists data to queues, tables, and blobs in an [Azure Storage](#) account that you specify.

Types of data that is serialized and persisted

The following is a list of the different types of data that will be serialized and persisted when using features of Durable Functions:

- All inputs and outputs of orchestrator, activity, and entity functions, including any IDs and unhandled exceptions
- Orchestrator, activity, and entity function names
- External event names and payloads
- Custom orchestration status payloads
- Orchestration termination messages
- Durable timer payloads
- Durable HTTP request and response URLs, headers, and payloads
- Entity call and signal payloads
- Entity state payloads

Working with sensitive data

When using Azure Storage, all data is automatically encrypted at rest. However, anyone with access to the storage account can read the data in its unencrypted form. If you need stronger protection for sensitive data, consider first encrypting the data using your own encryption keys so that Durable Functions persists the data in

a pre-encrypted form.

Alternatively, .NET users have the option of implementing custom serialization providers that provide automatic encryption. An example of custom serialization with encryption can be found in [this GitHub sample](#).

NOTE

If you decide to implement application-level encryption, be aware that orchestrations and entities can exist for indefinite amounts of time. This matters when it comes time to rotate your encryption keys because an orchestration or entities may run longer than your key rotation policy. If a key rotation happens, the key used to encrypt your data may no longer be available to decrypt it the next time your orchestration or entity executes. Customer encryption is therefore recommended only when orchestrations and entities are expected to run for relatively short periods of time.

Customizing serialization and deserialization

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

Default serialization logic

Durable Functions for .NET in-process internally uses [Json.NET](#) to serialize orchestration and entity data to JSON. The default settings Durable Functions uses for Json.NET are:

Inputs, Outputs, and State:

```
JsonSerializerSettings
{
    TypeNameHandling = TypeNameHandling.None,
    DateParseHandling = DateParseHandling.None,
}
```

Exceptions:

```
JsonSerializerSettings
{
    ContractResolver = new ExceptionResolver(),
    TypeNameHandling = TypeNameHandling.Objects,
    ReferenceLoopHandling = ReferenceLoopHandling.Ignore,
}
```

Read more detailed documentation about `JsonSerializerSettings` [here](#).

Customizing serialization with .NET attributes

When serializing data, Json.NET looks for [various attributes](#) on classes and properties that control how the data is serialized and deserialized from JSON. If you own the source code for data type passed to Durable Functions APIs, consider adding these attributes to the type to customize serialization and deserialization.

Customizing serialization with Dependency Injection

Function apps that target .NET and run on the Functions V3 runtime can use [Dependency Injection \(DI\)](#) to customize how data and exceptions are serialized. The sample code below demonstrates how to use DI to override the default Json.NET serialization settings using custom implementations of the

`IMessageSerializerSettingsFactory` and `IErrorSerializerSettingsFactory` service interfaces.

```
using Microsoft.Azure.Functions.Extensions.DependencyInjection;
using Microsoft.Azure.WebJobs.Extensions.DurableTask;
using Microsoft.Extensions.DependencyInjection;
using Newtonsoft.Json;
using System.Collections.Generic;

[assembly: FunctionsStartup(typeof(MyApplication.Startup))]
namespace MyApplication
{
    public class Startup : FunctionsStartup
    {
        public override void Configure(IFunctionsHostBuilder builder)
        {
            builder.Services.AddSingleton<IMessageSerializerSettingsFactory,
CustomMessageSerializerSettingsFactory>();
            builder.Services.AddSingleton<IErrorSerializerSettingsFactory,
CustomErrorSerializerSettingsFactory>();
        }

        /// <summary>
        /// A factory that provides the serialization for all inputs and outputs for activities and
        /// orchestrations, as well as entity state.
        /// </summary>
        internal class CustomMessageSerializerSettingsFactory : IMessageSerializerSettingsFactory
        {
            public JsonSerializerSettings CreateJsonSerializerSettings()
            {
                // Return your custom JsonSerializerSettings here
            }
        }

        /// <summary>
        /// A factory that provides the serialization for all exceptions thrown by activities
        /// and orchestrations
        /// </summary>
        internal class CustomErrorSerializerSettingsFactory : IErrorSerializerSettingsFactory
        {
            public JsonSerializerSettings CreateJsonSerializerSettings()
            {
                // Return your custom JsonSerializerSettings here
            }
        }
    }
}
```

.NET Isolated and System.Text.Json

Durable Functions running in the [.NET Isolated worker process](#) use `System.Text.Json` libraries for serialization rather than `Newtonsoft.Json`. There is currently no support for injecting serialization settings. However, attributes may be used to control aspects of serialization.

For more information on the built-in support for JSON serialization in .NET, see the [JSON serialization and deserialization in .NET overview documentation](#).

Bindings for Durable Functions (Azure Functions)

10/5/2022 • 17 minutes to read • [Edit Online](#)

The [Durable Functions](#) extension introduces three trigger bindings that control the execution of orchestrator, entity, and activity functions. It also introduces an output binding that acts as a client for the Durable Functions runtime.

Orchestration trigger

The orchestration trigger enables you to author [durable orchestrator functions](#). This trigger executes when a new orchestration instance is scheduled and when an existing orchestration instance receives an event.

Examples of events that can trigger orchestrator functions include durable timer expirations, activity function responses, and events raised by external clients.

When you author functions in .NET, the orchestration trigger is configured using the [OrchestrationTriggerAttribute](#) .NET attribute. For Java, the `@DurableOrchestrationTrigger` annotation is used.

When you write orchestrator functions in scripting languages, like JavaScript, Python, or PowerShell, the orchestration trigger is defined by the following JSON object in the `bindings` array of the `function.json` file:

```
{  
    "name": "<Name of input parameter in function signature>",  
    "orchestration": "<Optional - name of the orchestration>",  
    "type": "orchestrationTrigger",  
    "direction": "in"  
}
```

- `orchestration` is the name of the orchestration that clients must use when they want to start new instances of this orchestrator function. This property is optional. If not specified, the name of the function is used.

Internally, this trigger binding polls the configured durable store for new orchestration events, such as orchestration start events, durable timer expiration events, activity function response events, and external events raised by other functions.

Trigger behavior

Here are some notes about the orchestration trigger:

- **Single-threading** - A single dispatcher thread is used for all orchestrator function execution on a single host instance. For this reason, it's important to ensure that orchestrator function code is efficient and doesn't perform any I/O. It is also important to ensure that this thread does not do any async work except when awaiting on Durable Functions-specific task types.
- **Poison-message handling** - There's no poison message support in orchestration triggers.
- **Message visibility** - Orchestration trigger messages are dequeued and kept invisible for a configurable duration. The visibility of these messages is renewed automatically as long as the function app is running and healthy.
- **Return values** - Return values are serialized to JSON and persisted to the orchestration history table in Azure Table storage. These return values can be queried by the orchestration client binding, described later.

WARNING

Orchestrator functions should never use any input or output bindings other than the orchestration trigger binding. Doing so has the potential to cause problems with the Durable Task extension because those bindings may not obey the single-threading and I/O rules. If you'd like to use other bindings, add them to an activity function called from your orchestrator function. For more information about coding constraints for orchestrator functions, see the [Orchestrator function code constraints](#) documentation.

WARNING

JavaScript and Python orchestrator functions should never be declared `async`.

Trigger usage

The orchestration trigger binding supports both inputs and outputs. Here are some things to know about input and output handling:

- **inputs** - Orchestration triggers can be invoked with inputs, which are accessed through the context input object. All inputs must be JSON-serializable.
- **outputs** - Orchestration triggers support output values as well as inputs. The return value of the function is used to assign the output value and must be JSON-serializable.

Trigger sample

The following example code shows what the simplest "Hello World" orchestrator function might look like. Note that this example orchestrator doesn't actually schedule any tasks.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```
[FunctionName("HelloWorld")]
public static string Run([OrchestrationTrigger] IDurableOrchestrationContext context)
{
    string name = context.GetInput<string>();
    return $"Hello {name}!";
}
```

NOTE

The previous code is for Durable Functions 2.x. For Durable Functions 1.x, you must use `DurableOrchestrationContext` instead of `IDurableOrchestrationContext`. For more information about the differences between versions, see the [Durable Functions Versions](#) article.

Most orchestrator functions call activity functions, so here is a "Hello World" example that demonstrates how to call an activity function:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)

- Java

```
[FunctionName("HelloWorld")]
public static async Task<string> Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    string name = context.GetInput<string>();
    string result = await context.CallActivityAsync<string>("SayHello", name);
    return result;
}
```

NOTE

The previous code is for Durable Functions 2.x. For Durable Functions 1.x, you must use `DurableOrchestrationContext` instead of `IDurableOrchestrationContext`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

Activity trigger

The activity trigger enables you to author functions that are called by orchestrator functions, known as [activity functions](#).

If you're authoring functions in .NET, the activity trigger is configured using the [ActivityTriggerAttribute](#) .NET attribute. For Java, the `@DurableActivityTrigger` annotation is used.

If you're using JavaScript, Python, or PowerShell, the activity trigger is defined by the following JSON object in the `bindings` array of `function.json`:

```
{
    "name": "<Name of input parameter in function signature>",
    "activity": "<Optional - name of the activity>",
    "type": "activityTrigger",
    "direction": "in"
}
```

- `activity` is the name of the activity. This value is the name that orchestrator functions use to invoke this activity function. This property is optional. If not specified, the name of the function is used.

Internally, this trigger binding polls the configured durable store for new activity execution events.

Trigger behavior

Here are some notes about the activity trigger:

- **Threading** - Unlike the orchestration trigger, activity triggers don't have any restrictions around threading or I/O. They can be treated like regular functions.
- **Poison-message handling** - There's no poison message support in activity triggers.
- **Message visibility** - Activity trigger messages are dequeued and kept invisible for a configurable duration. The visibility of these messages is renewed automatically as long as the function app is running and healthy.
- **Return values** - Return values are serialized to JSON and persisted to the configured durable store.

Trigger usage

The activity trigger binding supports both inputs and outputs, just like the orchestration trigger. Here are some things to know about input and output handling:

- **inputs** - Activity triggers can be invoked with inputs from an orchestrator function. All inputs must be JSON-

serializable.

- **outputs** - Activity functions support output values as well as inputs. The return value of the function is used to assign the output value and must be JSON-serializable.
- **metadata** - .NET activity functions can bind to a `string instanceId` parameter to get the instance ID of the calling orchestration.

Trigger sample

The following example code shows what a simple `SayHello` activity function might look like:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```
[FunctionName("SayHello")]
public static string SayHello([ActivityTrigger] IDurableActivityContext helloContext)
{
    string name = helloContext.GetInput<string>();
    return $"Hello {name}!";
}
```

The default parameter type for the .NET `ActivityTriggerAttribute` binding is `IDurableActivityContext` (or `DurableActivityContext` for Durable Functions v1). However, .NET activity triggers also support binding directly to JSON-serializable types (including primitive types), so the same function could be simplified as follows:

```
[FunctionName("SayHello")]
public static string SayHello([ActivityTrigger] string name)
{
    return $"Hello {name}!";
}
```

Using input and output bindings

You can use regular input and output bindings in addition to the activity trigger binding. For example, you can take the input to your activity binding, and send a message to an EventHub using the EventHub output binding:

```
{
  "bindings": [
    {
      "name": "message",
      "type": "activityTrigger",
      "direction": "in"
    },
    {
      "type": "eventHub",
      "name": "outputEventHubMessage",
      "connection": "EventhubConnectionSetting",
      "eventHubName": "eh_messages",
      "direction": "out"
    }
  ]
}
```

```
module.exports = async function (context) {
    context.bindings.outputEventHubMessage = context.bindings.message;
};
```

Orchestration client

The orchestration client binding enables you to write functions that interact with orchestrator functions. These functions are often referred to as [client functions](#). For example, you can act on orchestration instances in the following ways:

- Start them.
- Query their status.
- Terminate them.
- Send events to them while they're running.
- Purge instance history.

If you're using .NET, you can bind to the orchestration client by using the [DurableClientAttribute](#) attribute ([OrchestrationClientAttribute](#) in Durable Functions v1.x). For Java, use the `@DurableClientInput` annotation.

If you're using scripting languages, like JavaScript, Python, or PowerShell, the durable client trigger is defined by the following JSON object in the `bindings` array of *function.json*.

```
{
    "name": "<Name of input parameter in function signature>",
    "taskHub": "<Optional - name of the task hub>",
    "connectionName": "<Optional - name of the connection string app setting>",
    "type": "orchestrationClient",
    "direction": "in"
}
```

- `taskHub` - Used in scenarios where multiple function apps share the same storage account but need to be isolated from each other. If not specified, the default value from `host.json` is used. This value must match the value used by the target orchestrator functions.
- `connectionName` - The name of an app setting that contains a storage account connection string. The storage account represented by this connection string must be the same one used by the target orchestrator functions. If not specified, the default storage account connection string for the function app is used.

NOTE

In most cases, we recommend that you omit these properties and rely on the default behavior.

Client usage

In .NET functions, you typically bind to [IDurableClient](#) ([DurableOrchestrationClient](#) in Durable Functions v1.x), which gives you full access to all orchestration client APIs supported by Durable Functions. For Java, you bind to the `DurableClientContext` class. In other languages, you must use the language-specific SDK to get access to a client object.

Here's an example queue-triggered function that starts a "HelloWorld" orchestration.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)

- Java

```
[FunctionName("QueueStart")]
public static Task Run(
    [QueueTrigger("durable-function-trigger")] string input,
    [DurableClient] IDurableOrchestrationClient starter)
{
    // Orchestration input comes from the queue message content.
    return starter.StartNewAsync("HelloWorld", input);
}
```

NOTE

The previous C# code is for Durable Functions 2.x. For Durable Functions 1.x, you must use `OrchestrationClient` attribute instead of the `DurableClient` attribute, and you must use the `DurableOrchestrationClient` parameter type instead of `IDurableOrchestrationClient`. For more information about the differences between versions, see the [Durable Functions Versions](#) article.

More details on starting instances can be found in [Instance management](#).

Entity trigger

Entity triggers allow you to author [entity functions](#). This trigger supports processing events for a specific entity instance.

NOTE

Entity triggers are available starting in Durable Functions 2.x.

Internally, this trigger binding polls the configured durable store for new entity operations that need to be executed.

If you're authoring functions in .NET, the entity trigger is configured using the [EntityTriggerAttribute](#) .NET attribute.

If you're using JavaScript, Python, or PowerShell, the entity trigger is defined by the following JSON object in the `bindings` array of `function.json`:

```
{
    "name": "<Name of input parameter in function signature>",
    "entityName": "<Optional - name of the entity>",
    "type": "entityTrigger",
    "direction": "in"
}
```

NOTE

Entity triggers are not yet supported in Java.

By default, the name of an entity is the name of the function.

Trigger behavior

Here are some notes about the entity trigger:

- **Single-threaded**: A single dispatcher thread is used to process operations for a particular entity. If multiple messages are sent to a single entity concurrently, the operations will be processed one-at-a-time.
- **Poison-message handling** - There's no poison message support in entity triggers.
- **Message visibility** - Entity trigger messages are dequeued and kept invisible for a configurable duration. The visibility of these messages is renewed automatically as long as the function app is running and healthy.
- **Return values** - Entity functions don't support return values. There are specific APIs that can be used to save state or pass values back to orchestrations.

Any state changes made to an entity during its execution will be automatically persisted after execution has completed.

For more information and examples on defining and interacting with entity triggers, see the [Durable Entities](#) documentation.

Entity client

The entity client binding enables you to asynchronously trigger [entity functions](#). These functions are sometimes referred to as [client functions](#).

If you're using .NET precompiled functions, you can bind to the entity client by using the [DurableClientAttribute](#) .NET attribute.

NOTE

The `[DurableClientAttribute]` can also be used to bind to the [orchestration client](#).

If you're using scripting languages (like C# scripting, JavaScript, or Python) for development, the entity trigger is defined by the following JSON object in the `bindings` array of *function.json*:

```
{
  "name": "<Name of input parameter in function signature>",
  "taskHub": "<Optional - name of the task hub>",
  "connectionName": "<Optional - name of the connection string app setting>",
  "type": "durableClient",
  "direction": "in"
}
```

NOTE

Entity clients are not yet supported in Java.

- `taskHub` - Used in scenarios where multiple function apps share the same storage account but need to be isolated from each other. If not specified, the default value from `host.json` is used. This value must match the value used by the target entity functions.
- `connectionName` - The name of an app setting that contains a storage account connection string. The storage account represented by this connection string must be the same one used by the target entity functions. If not specified, the default storage account connection string for the function app is used.

NOTE

In most cases, we recommend that you omit the optional properties and rely on the default behavior.

For more information and examples on interacting with entities as a client, see the [Durable Entities](#)

documentation.

host.json settings

Configuration settings for [Durable Functions](#).

NOTE

All major versions of Durable Functions are supported on all versions of the Azure Functions runtime. However, the schema of the host.json configuration is slightly different depending on the version of the Azure Functions runtime and the Durable Functions extension version you use. The following examples are for use with Azure Functions 2.0 and 3.0. In both examples, if you're using Azure Functions 1.0, the available settings are the same, but the "durableTask" section of the host.json should go in the root of the host.json configuration instead of as a field under "extensions".

- [Durable Functions 2.x](#)
- [Durable Functions 1.x](#)

```
{
  "extensions": {
    "durableTask": {
      "hubName": "MyTaskHub",
      "storageProvider": {
        "connectionStringName": "AzureWebJobsStorage",
        "controlQueueBatchSize": 32,
        "controlQueueBufferThreshold": 256,
        "controlQueueVisibilityTimeout": "00:05:00",
        "maxQueuePollingInterval": "00:00:30",
        "partitionCount": 4,
        "trackingStoreConnectionStringName": "TrackingStorage",
        "trackingStoreNamePrefix": "DurableTask",
        "useLegacyPartitionManagement": true,
        "workItemQueueVisibilityTimeout": "00:05:00"
      },
      "tracing": {
        "traceInputsAndOutputs": false,
        "traceReplayEvents": false
      },
      "notifications": {
        "eventGrid": {
          "topicEndpoint": "https://topic_name.westus2-1.eventgrid.azure.net/api/events",
          "keySettingName": "EventGridKey",
          "publishRetryCount": 3,
          "publishRetryInterval": "00:00:30",
          "publishEventTypes": [
            "Started",
            "Pending",
            "Failed",
            "Terminated"
          ]
        }
      },
      "maxConcurrentActivityFunctions": 10,
      "maxConcurrentOrchestratorFunctions": 10,
      "extendedSessionsEnabled": false,
      "extendedSessionIdleTimeoutInSeconds": 30,
      "useAppLease": true,
      "useGracefulShutdown": false,
      "maxEntityOperationBatchSize": 50
    }
  }
}
```

Task hub names must start with a letter and consist of only letters and numbers. If not specified, the default task hub name for a function app is **DurableFunctionsHub**. For more information, see [Task hubs](#).

PROPERTY	DEFAULT	DESCRIPTION
hubName	DurableFunctionsHub	Alternate task hub names can be used to isolate multiple Durable Functions applications from each other, even if they're using the same storage backend.
controlQueueBatchSize	32	The number of messages to pull from the control queue at a time.

PROPERTY	DEFAULT	DESCRIPTION
controlQueueBufferThreshold	Consumption plan for Python: 32 Consumption plan for JavaScript and C#: 128 Dedicated/Premium plan: 256	The number of control queue messages that can be buffered in memory at a time, at which point the dispatcher will wait before dequeuing any additional messages.
partitionCount	4	The partition count for the control queue. May be a positive integer between 1 and 16.
controlQueueVisibilityTimeout	5 minutes	The visibility timeout of dequeued control queue messages.
workItemQueueVisibilityTimeout	5 minutes	The visibility timeout of dequeued work item queue messages.
maxConcurrentActivityFunctions	Consumption plan: 10 Dedicated/Premium plan: 10X the number of processors on the current machine	The maximum number of activity functions that can be processed concurrently on a single host instance.
maxConcurrentOrchestratorFunctions	Consumption plan: 5 Dedicated/Premium plan: 10X the number of processors on the current machine	The maximum number of orchestrator functions that can be processed concurrently on a single host instance.
maxQueuePollingInterval	30 seconds	The maximum control and work-item queue polling interval in the <i>hh:mm:ss</i> format. Higher values can result in higher message processing latencies. Lower values can result in higher storage costs because of increased storage transactions.
connectionName (2.7.0 and later) connectionStringName (2.x) azureStorageConnectionStringName (1.x)	AzureWebJobsStorage	The name of an app setting or setting collection that specifies how to connect to the underlying Azure Storage resources. When a single app setting is provided, it should be an Azure Storage connection string.
trackingStoreConnectionName (2.7.0 and later) trackingStoreConnectionStringName		The name of an app setting or setting collection that specifies how to connect to the History and Instances tables. When a single app setting is provided, it should be an Azure Storage connection string. If not specified, the <code>connectionStringName</code> (Durable 2.x) or <code>azureStorageConnectionStringName</code> (Durable 1.x) connection is used.

PROPERTY	DEFAULT	DESCRIPTION
trackingStoreNamePrefix		<p>The prefix to use for the History and Instances tables when <code>trackingStoreConnectionStringName</code> is specified. If not set, the default prefix value will be <code>DurableTask</code>. If <code>trackingStoreConnectionStringName</code> is not specified, then the History and Instances tables will use the <code>hubName</code> value as their prefix, and any setting for <code>trackingStoreNamePrefix</code> will be ignored.</p>
traceInputsAndOutputs	false	<p>A value indicating whether to trace the inputs and outputs of function calls. The default behavior when tracing function execution events is to include the number of bytes in the serialized inputs and outputs for function calls. This behavior provides minimal information about what the inputs and outputs look like without bloating the logs or inadvertently exposing sensitive information. Setting this property to true causes the default function logging to log the entire contents of function inputs and outputs.</p>
traceReplayEvents	false	<p>A value indicating whether to write orchestration replay events to Application Insights.</p>
eventGridTopicEndpoint		<p>The URL of an Azure Event Grid custom topic endpoint. When this property is set, orchestration life-cycle notification events are published to this endpoint. This property supports App Settings resolution.</p>
eventGridKeySettingName		<p>The name of the app setting containing the key used for authenticating with the Azure Event Grid custom topic at <code>EventGridTopicEndpoint</code>.</p>
eventGridPublishRetryCount	0	<p>The number of times to retry if publishing to the Event Grid Topic fails.</p>
eventGridPublishRetryInterval	5 minutes	<p>The Event Grid publishes retry interval in the <i>hh:mm:ss</i> format.</p>
eventGridPublishEventTypes		<p>A list of event types to publish to Event Grid. If not specified, all event types will be published. Allowed values include <code>Started</code>, <code>Completed</code>, <code>Failed</code>, <code>Terminated</code>.</p>

PROPERTY	DEFAULT	DESCRIPTION
useAppLease	true	When set to <code>true</code> , apps will require acquiring an app-level blob lease before processing task hub messages. For more information, see the disaster recovery and geo-distribution documentation. Available starting in v2.3.0.
useLegacyPartitionManagement	false	When set to <code>false</code> , uses a partition management algorithm that reduces the possibility of duplicate function execution when scaling out. Available starting in v2.3.0.
useGracefulShutdown	false	(Preview) Enable gracefully shutting down to reduce the chance of host shutdowns failing in-process function executions.
maxEntityOperationBatchSize(2.6.1)	Consumption plan: 50 Dedicated/Premium plan: 5000	The maximum number of entity operations that are processed as a batch . If set to 1, batching is disabled, and each operation message is processed by a separate function invocation.

Many of these settings are for optimizing performance. For more information, see [Performance and scale](#).

Next steps

[Built-in HTTP API reference for instance management](#)

Bindings for Durable Functions (Azure Functions)

10/5/2022 • 17 minutes to read • [Edit Online](#)

The [Durable Functions](#) extension introduces three trigger bindings that control the execution of orchestrator, entity, and activity functions. It also introduces an output binding that acts as a client for the Durable Functions runtime.

Orchestration trigger

The orchestration trigger enables you to author [durable orchestrator functions](#). This trigger executes when a new orchestration instance is scheduled and when an existing orchestration instance receives an event.

Examples of events that can trigger orchestrator functions include durable timer expirations, activity function responses, and events raised by external clients.

When you author functions in .NET, the orchestration trigger is configured using the [OrchestrationTriggerAttribute](#) .NET attribute. For Java, the `@DurableOrchestrationTrigger` annotation is used.

When you write orchestrator functions in scripting languages, like JavaScript, Python, or PowerShell, the orchestration trigger is defined by the following JSON object in the `bindings` array of the `function.json` file:

```
{  
    "name": "<Name of input parameter in function signature>",  
    "orchestration": "<Optional - name of the orchestration>",  
    "type": "orchestrationTrigger",  
    "direction": "in"  
}
```

- `orchestration` is the name of the orchestration that clients must use when they want to start new instances of this orchestrator function. This property is optional. If not specified, the name of the function is used.

Internally, this trigger binding polls the configured durable store for new orchestration events, such as orchestration start events, durable timer expiration events, activity function response events, and external events raised by other functions.

Trigger behavior

Here are some notes about the orchestration trigger:

- **Single-threading** - A single dispatcher thread is used for all orchestrator function execution on a single host instance. For this reason, it's important to ensure that orchestrator function code is efficient and doesn't perform any I/O. It is also important to ensure that this thread does not do any async work except when awaiting on Durable Functions-specific task types.
- **Poison-message handling** - There's no poison message support in orchestration triggers.
- **Message visibility** - Orchestration trigger messages are dequeued and kept invisible for a configurable duration. The visibility of these messages is renewed automatically as long as the function app is running and healthy.
- **Return values** - Return values are serialized to JSON and persisted to the orchestration history table in Azure Table storage. These return values can be queried by the orchestration client binding, described later.

WARNING

Orchestrator functions should never use any input or output bindings other than the orchestration trigger binding. Doing so has the potential to cause problems with the Durable Task extension because those bindings may not obey the single-threading and I/O rules. If you'd like to use other bindings, add them to an activity function called from your orchestrator function. For more information about coding constraints for orchestrator functions, see the [Orchestrator function code constraints](#) documentation.

WARNING

JavaScript and Python orchestrator functions should never be declared `async`.

Trigger usage

The orchestration trigger binding supports both inputs and outputs. Here are some things to know about input and output handling:

- **inputs** - Orchestration triggers can be invoked with inputs, which are accessed through the context input object. All inputs must be JSON-serializable.
- **outputs** - Orchestration triggers support output values as well as inputs. The return value of the function is used to assign the output value and must be JSON-serializable.

Trigger sample

The following example code shows what the simplest "Hello World" orchestrator function might look like. Note that this example orchestrator doesn't actually schedule any tasks.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```
[FunctionName("HelloWorld")]
public static string Run([OrchestrationTrigger] IDurableOrchestrationContext context)
{
    string name = context.GetInput<string>();
    return $"Hello {name}!";
}
```

NOTE

The previous code is for Durable Functions 2.x. For Durable Functions 1.x, you must use `DurableOrchestrationContext` instead of `IDurableOrchestrationContext`. For more information about the differences between versions, see the [Durable Functions Versions](#) article.

Most orchestrator functions call activity functions, so here is a "Hello World" example that demonstrates how to call an activity function:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)

- Java

```
[FunctionName("HelloWorld")]
public static async Task<string> Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    string name = context.GetInput<string>();
    string result = await context.CallActivityAsync<string>("SayHello", name);
    return result;
}
```

NOTE

The previous code is for Durable Functions 2.x. For Durable Functions 1.x, you must use `DurableOrchestrationContext` instead of `IDurableOrchestrationContext`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

Activity trigger

The activity trigger enables you to author functions that are called by orchestrator functions, known as [activity functions](#).

If you're authoring functions in .NET, the activity trigger is configured using the [ActivityTriggerAttribute](#) .NET attribute. For Java, the `@DurableActivityTrigger` annotation is used.

If you're using JavaScript, Python, or PowerShell, the activity trigger is defined by the following JSON object in the `bindings` array of `function.json`:

```
{
    "name": "<Name of input parameter in function signature>",
    "activity": "<Optional - name of the activity>",
    "type": "activityTrigger",
    "direction": "in"
}
```

- `activity` is the name of the activity. This value is the name that orchestrator functions use to invoke this activity function. This property is optional. If not specified, the name of the function is used.

Internally, this trigger binding polls the configured durable store for new activity execution events.

Trigger behavior

Here are some notes about the activity trigger:

- **Threading** - Unlike the orchestration trigger, activity triggers don't have any restrictions around threading or I/O. They can be treated like regular functions.
- **Poison-message handling** - There's no poison message support in activity triggers.
- **Message visibility** - Activity trigger messages are dequeued and kept invisible for a configurable duration. The visibility of these messages is renewed automatically as long as the function app is running and healthy.
- **Return values** - Return values are serialized to JSON and persisted to the configured durable store.

Trigger usage

The activity trigger binding supports both inputs and outputs, just like the orchestration trigger. Here are some things to know about input and output handling:

- **inputs** - Activity triggers can be invoked with inputs from an orchestrator function. All inputs must be JSON-

serializable.

- **outputs** - Activity functions support output values as well as inputs. The return value of the function is used to assign the output value and must be JSON-serializable.
- **metadata** - .NET activity functions can bind to a `string instanceId` parameter to get the instance ID of the calling orchestration.

Trigger sample

The following example code shows what a simple `SayHello` activity function might look like:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)
- [Java](#)

```
[FunctionName("SayHello")]
public static string SayHello([ActivityTrigger] IDurableActivityContext helloContext)
{
    string name = helloContext.GetInput<string>();
    return $"Hello {name}!";
}
```

The default parameter type for the .NET `ActivityTriggerAttribute` binding is `IDurableActivityContext` (or `DurableActivityContext` for Durable Functions v1). However, .NET activity triggers also support binding directly to JSON-serializable types (including primitive types), so the same function could be simplified as follows:

```
[FunctionName("SayHello")]
public static string SayHello([ActivityTrigger] string name)
{
    return $"Hello {name}!";
}
```

Using input and output bindings

You can use regular input and output bindings in addition to the activity trigger binding. For example, you can take the input to your activity binding, and send a message to an EventHub using the EventHub output binding:

```
{
  "bindings": [
    {
      "name": "message",
      "type": "activityTrigger",
      "direction": "in"
    },
    {
      "type": "eventHub",
      "name": "outputEventHubMessage",
      "connection": "EventhubConnectionSetting",
      "eventHubName": "eh_messages",
      "direction": "out"
    }
  ]
}
```

```
module.exports = async function (context) {
    context.bindings.outputEventHubMessage = context.bindings.message;
};
```

Orchestration client

The orchestration client binding enables you to write functions that interact with orchestrator functions. These functions are often referred to as [client functions](#). For example, you can act on orchestration instances in the following ways:

- Start them.
- Query their status.
- Terminate them.
- Send events to them while they're running.
- Purge instance history.

If you're using .NET, you can bind to the orchestration client by using the [DurableClientAttribute](#) attribute ([OrchestrationClientAttribute](#) in Durable Functions v1.x). For Java, use the `@DurableClientInput` annotation.

If you're using scripting languages, like JavaScript, Python, or PowerShell, the durable client trigger is defined by the following JSON object in the `bindings` array of *function.json*.

```
{
    "name": "<Name of input parameter in function signature>",
    "taskHub": "<Optional - name of the task hub>",
    "connectionName": "<Optional - name of the connection string app setting>",
    "type": "orchestrationClient",
    "direction": "in"
}
```

- `taskHub` - Used in scenarios where multiple function apps share the same storage account but need to be isolated from each other. If not specified, the default value from `host.json` is used. This value must match the value used by the target orchestrator functions.
- `connectionName` - The name of an app setting that contains a storage account connection string. The storage account represented by this connection string must be the same one used by the target orchestrator functions. If not specified, the default storage account connection string for the function app is used.

NOTE

In most cases, we recommend that you omit these properties and rely on the default behavior.

Client usage

In .NET functions, you typically bind to [IDurableClient](#) ([DurableOrchestrationClient](#) in Durable Functions v1.x), which gives you full access to all orchestration client APIs supported by Durable Functions. For Java, you bind to the `DurableClientContext` class. In other languages, you must use the language-specific SDK to get access to a client object.

Here's an example queue-triggered function that starts a "HelloWorld" orchestration.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [PowerShell](#)

- Java

```
[FunctionName("QueueStart")]
public static Task Run(
    [QueueTrigger("durable-function-trigger")] string input,
    [DurableClient] IDurableOrchestrationClient starter)
{
    // Orchestration input comes from the queue message content.
    return starter.StartNewAsync("HelloWorld", input);
}
```

NOTE

The previous C# code is for Durable Functions 2.x. For Durable Functions 1.x, you must use `OrchestrationClient` attribute instead of the `DurableClient` attribute, and you must use the `DurableOrchestrationClient` parameter type instead of `IDurableOrchestrationClient`. For more information about the differences between versions, see the [Durable Functions Versions](#) article.

More details on starting instances can be found in [Instance management](#).

Entity trigger

Entity triggers allow you to author [entity functions](#). This trigger supports processing events for a specific entity instance.

NOTE

Entity triggers are available starting in Durable Functions 2.x.

Internally, this trigger binding polls the configured durable store for new entity operations that need to be executed.

If you're authoring functions in .NET, the entity trigger is configured using the [EntityTriggerAttribute](#) .NET attribute.

If you're using JavaScript, Python, or PowerShell, the entity trigger is defined by the following JSON object in the `bindings` array of `function.json`:

```
{
    "name": "<Name of input parameter in function signature>",
    "entityName": "<Optional - name of the entity>",
    "type": "entityTrigger",
    "direction": "in"
}
```

NOTE

Entity triggers are not yet supported in Java.

By default, the name of an entity is the name of the function.

Trigger behavior

Here are some notes about the entity trigger:

- **Single-threaded**: A single dispatcher thread is used to process operations for a particular entity. If multiple messages are sent to a single entity concurrently, the operations will be processed one-at-a-time.
- **Poison-message handling** - There's no poison message support in entity triggers.
- **Message visibility** - Entity trigger messages are dequeued and kept invisible for a configurable duration. The visibility of these messages is renewed automatically as long as the function app is running and healthy.
- **Return values** - Entity functions don't support return values. There are specific APIs that can be used to save state or pass values back to orchestrations.

Any state changes made to an entity during its execution will be automatically persisted after execution has completed.

For more information and examples on defining and interacting with entity triggers, see the [Durable Entities](#) documentation.

Entity client

The entity client binding enables you to asynchronously trigger [entity functions](#). These functions are sometimes referred to as [client functions](#).

If you're using .NET precompiled functions, you can bind to the entity client by using the [DurableClientAttribute](#) .NET attribute.

NOTE

The `[DurableClientAttribute]` can also be used to bind to the [orchestration client](#).

If you're using scripting languages (like C# scripting, JavaScript, or Python) for development, the entity trigger is defined by the following JSON object in the `bindings` array of `function.json`:

```
{
  "name": "<Name of input parameter in function signature>",
  "taskHub": "<Optional - name of the task hub>",
  "connectionName": "<Optional - name of the connection string app setting>",
  "type": "durableClient",
  "direction": "in"
}
```

NOTE

Entity clients are not yet supported in Java.

- `taskHub` - Used in scenarios where multiple function apps share the same storage account but need to be isolated from each other. If not specified, the default value from `host.json` is used. This value must match the value used by the target entity functions.
- `connectionName` - The name of an app setting that contains a storage account connection string. The storage account represented by this connection string must be the same one used by the target entity functions. If not specified, the default storage account connection string for the function app is used.

NOTE

In most cases, we recommend that you omit the optional properties and rely on the default behavior.

For more information and examples on interacting with entities as a client, see the [Durable Entities](#)

documentation.

host.json settings

Configuration settings for [Durable Functions](#).

NOTE

All major versions of Durable Functions are supported on all versions of the Azure Functions runtime. However, the schema of the host.json configuration is slightly different depending on the version of the Azure Functions runtime and the Durable Functions extension version you use. The following examples are for use with Azure Functions 2.0 and 3.0. In both examples, if you're using Azure Functions 1.0, the available settings are the same, but the "durableTask" section of the host.json should go in the root of the host.json configuration instead of as a field under "extensions".

- [Durable Functions 2.x](#)
- [Durable Functions 1.x](#)

```
{
  "extensions": {
    "durableTask": {
      "hubName": "MyTaskHub",
      "storageProvider": {
        "connectionStringName": "AzureWebJobsStorage",
        "controlQueueBatchSize": 32,
        "controlQueueBufferThreshold": 256,
        "controlQueueVisibilityTimeout": "00:05:00",
        "maxQueuePollingInterval": "00:00:30",
        "partitionCount": 4,
        "trackingStoreConnectionStringName": "TrackingStorage",
        "trackingStoreNamePrefix": "DurableTask",
        "useLegacyPartitionManagement": true,
        "workItemQueueVisibilityTimeout": "00:05:00"
      },
      "tracing": {
        "traceInputsAndOutputs": false,
        "traceReplayEvents": false
      },
      "notifications": {
        "eventGrid": {
          "topicEndpoint": "https://topic_name.westus2-1.eventgrid.azure.net/api/events",
          "keySettingName": "EventGridKey",
          "publishRetryCount": 3,
          "publishRetryInterval": "00:00:30",
          "publishEventTypes": [
            "Started",
            "Pending",
            "Failed",
            "Terminated"
          ]
        }
      },
      "maxConcurrentActivityFunctions": 10,
      "maxConcurrentOrchestratorFunctions": 10,
      "extendedSessionsEnabled": false,
      "extendedSessionIdleTimeoutInSeconds": 30,
      "useAppLease": true,
      "useGracefulShutdown": false,
      "maxEntityOperationBatchSize": 50
    }
  }
}
```

Task hub names must start with a letter and consist of only letters and numbers. If not specified, the default task hub name for a function app is **DurableFunctionsHub**. For more information, see [Task hubs](#).

PROPERTY	DEFAULT	DESCRIPTION
hubName	DurableFunctionsHub	Alternate task hub names can be used to isolate multiple Durable Functions applications from each other, even if they're using the same storage backend.
controlQueueBatchSize	32	The number of messages to pull from the control queue at a time.

PROPERTY	DEFAULT	DESCRIPTION
controlQueueBufferThreshold	Consumption plan for Python: 32 Consumption plan for JavaScript and C#: 128 Dedicated/Premium plan: 256	The number of control queue messages that can be buffered in memory at a time, at which point the dispatcher will wait before dequeuing any additional messages.
partitionCount	4	The partition count for the control queue. May be a positive integer between 1 and 16.
controlQueueVisibilityTimeout	5 minutes	The visibility timeout of dequeued control queue messages.
workItemQueueVisibilityTimeout	5 minutes	The visibility timeout of dequeued work item queue messages.
maxConcurrentActivityFunctions	Consumption plan: 10 Dedicated/Premium plan: 10X the number of processors on the current machine	The maximum number of activity functions that can be processed concurrently on a single host instance.
maxConcurrentOrchestratorFunctions	Consumption plan: 5 Dedicated/Premium plan: 10X the number of processors on the current machine	The maximum number of orchestrator functions that can be processed concurrently on a single host instance.
maxQueuePollingInterval	30 seconds	The maximum control and work-item queue polling interval in the <i>hh:mm:ss</i> format. Higher values can result in higher message processing latencies. Lower values can result in higher storage costs because of increased storage transactions.
connectionName (2.7.0 and later) connectionStringName (2.x) azureStorageConnectionStringName (1.x)	AzureWebJobsStorage	The name of an app setting or setting collection that specifies how to connect to the underlying Azure Storage resources. When a single app setting is provided, it should be an Azure Storage connection string.
trackingStoreConnectionName (2.7.0 and later) trackingStoreConnectionStringName		The name of an app setting or setting collection that specifies how to connect to the History and Instances tables. When a single app setting is provided, it should be an Azure Storage connection string. If not specified, the <code>connectionStringName</code> (Durable 2.x) or <code>azureStorageConnectionStringName</code> (Durable 1.x) connection is used.

PROPERTY	DEFAULT	DESCRIPTION
trackingStoreNamePrefix		<p>The prefix to use for the History and Instances tables when <code>trackingStoreConnectionStringName</code> is specified. If not set, the default prefix value will be <code>DurableTask</code>. If <code>trackingStoreConnectionStringName</code> is not specified, then the History and Instances tables will use the <code>hubName</code> value as their prefix, and any setting for <code>trackingStoreNamePrefix</code> will be ignored.</p>
traceInputsAndOutputs	false	<p>A value indicating whether to trace the inputs and outputs of function calls. The default behavior when tracing function execution events is to include the number of bytes in the serialized inputs and outputs for function calls. This behavior provides minimal information about what the inputs and outputs look like without bloating the logs or inadvertently exposing sensitive information. Setting this property to true causes the default function logging to log the entire contents of function inputs and outputs.</p>
traceReplayEvents	false	<p>A value indicating whether to write orchestration replay events to Application Insights.</p>
eventGridTopicEndpoint		<p>The URL of an Azure Event Grid custom topic endpoint. When this property is set, orchestration life-cycle notification events are published to this endpoint. This property supports App Settings resolution.</p>
eventGridKeySettingName		<p>The name of the app setting containing the key used for authenticating with the Azure Event Grid custom topic at <code>EventGridTopicEndpoint</code>.</p>
eventGridPublishRetryCount	0	<p>The number of times to retry if publishing to the Event Grid Topic fails.</p>
eventGridPublishRetryInterval	5 minutes	<p>The Event Grid publishes retry interval in the <i>hh:mm:ss</i> format.</p>
eventGridPublishEventTypes		<p>A list of event types to publish to Event Grid. If not specified, all event types will be published. Allowed values include <code>Started</code>, <code>Completed</code>, <code>Failed</code>, <code>Terminated</code>.</p>

PROPERTY	DEFAULT	DESCRIPTION
useAppLease	true	When set to <code>true</code> , apps will require acquiring an app-level blob lease before processing task hub messages. For more information, see the disaster recovery and geo-distribution documentation. Available starting in v2.3.0.
useLegacyPartitionManagement	false	When set to <code>false</code> , uses a partition management algorithm that reduces the possibility of duplicate function execution when scaling out. Available starting in v2.3.0.
useGracefulShutdown	false	(Preview) Enable gracefully shutting down to reduce the chance of host shutdowns failing in-process function executions.
maxEntityOperationBatchSize(2.6.1)	Consumption plan: 50 Dedicated/Premium plan: 5000	The maximum number of entity operations that are processed as a batch . If set to 1, batching is disabled, and each operation message is processed by a separate function invocation.

Many of these settings are for optimizing performance. For more information, see [Performance and scale](#).

Next steps

[Built-in HTTP API reference for instance management](#)

HTTP API reference

10/5/2022 • 19 minutes to read • [Edit Online](#)

The Durable Functions extension exposes a set of built-in HTTP APIs that can be used to perform management tasks on [orchestrations](#), [entities](#), and [task hubs](#). These HTTP APIs are extensibility webhooks that are authorized by the Azure Functions host but handled directly by the Durable Functions extension.

The base URL for the APIs mentioned in this article is the same as the base URL for your function app. When developing locally using the [Azure Functions Core Tools](#), the base URL is typically `http://localhost:7071`. In the Azure Functions hosted service, the base URL is typically `https://<appName>.azurewebsites.net`. Custom hostnames are also supported if configured on your App Service app.

All HTTP APIs implemented by the extension require the following parameters. The data type of all parameters is `string`.

PARAMETER	PARAMETER TYPE	DESCRIPTION
<code>taskHub</code>	Query string	The name of the task hub . If not specified, the current function app's task hub name is assumed.
<code>connection</code>	Query string	The name of the connection app setting for the backend storage provider. If not specified, the default connection configuration for the function app is assumed.
<code>systemKey</code>	Query string	The authorization key required to invoke the API.

`systemKey` is an authorization key autogenerated by the Azure Functions host. It specifically grants access to the Durable Task extension APIs and can be managed the same way as [other Azure Functions access keys](#). You can generate URLs that contain the correct `taskHub`, `connection`, and `systemKey` query string values using [orchestration client binding](#) APIs, such as the `CreateCheckStatusResponse` and `CreateHttpManagementPayload` APIs in .NET, the `createCheckStatusResponse` and `createHttpManagementPayload` APIs in JavaScript, etc.

The next few sections cover the specific HTTP APIs supported by the extension and provide examples of how they can be used.

Start orchestration

Starts executing a new instance of the specified orchestrator function.

Request

For version 1.x of the Functions runtime, the request is formatted as follows (multiple lines are shown for clarity):

```
POST /admin/extensions/DurableTaskExtension/orchestrators/{functionName}/{instanceId?}
?taskHub={taskHub}
&connection={connectionName}
&code={systemKey}
```

In version 2.x of the Functions runtime, the URL format has all the same parameters but with a slightly different prefix:

```
POST /runtime/webhooks/durabletask/orchestrators/{functionName}/{instanceId?}  
    ?taskHub={taskHub}  
    &connection={connectionName}  
    &code={systemKey}
```

Request parameters for this API include the default set mentioned previously as well as the following unique parameters:

FIELD	PARAMETER TYPE	DESCRIPTION
<code>functionName</code>	URL	The name of the orchestrator function to start.
<code>instanceId</code>	URL	Optional parameter. The ID of the orchestration instance. If not specified, the orchestrator function will start with a random instance ID.
<code>{content}</code>	Request content	Optional. The JSON-formatted orchestrator function input.

Response

Several possible status code values can be returned.

- **HTTP 202 (Accepted)**: The specified orchestrator function was scheduled to start running. The `Location` response header contains a URL for polling the orchestration status.
- **HTTP 400 (Bad request)**: The specified orchestrator function doesn't exist, the specified instance ID was not valid, or request content was not valid JSON.

The following is an example request that starts a `RestartVMs` orchestrator function and includes JSON object payload:

```
POST /runtime/webhooks/durabletask/orchestrators/RestartVMs?code=XXX  
Content-Type: application/json  
Content-Length: 83  
  
{  
    "resourceGroup": "myRG",  
    "subscriptionId": "111deb5d-09df-4604-992e-a968345530a9"  
}
```

The response payload for the **HTTP 202** cases is a JSON object with the following fields:

FIELD	DESCRIPTION
<code>id</code>	The ID of the orchestration instance.
<code>statusQueryGetUri</code>	The status URL of the orchestration instance.
<code>sendEventPostUri</code>	The "raise event" URL of the orchestration instance.
<code>terminatePostUri</code>	The "terminate" URL of the orchestration instance.

FIELD	DESCRIPTION
<code>purgeHistoryDeleteUri</code>	The "purge history" URL of the orchestration instance.
<code>rewindPostUri</code>	(preview) The "rewind" URL of the orchestration instance.
<code>suspendPostUri</code>	The "suspend" URL of the orchestration instance.
<code>resumePostUri</code>	The "resume" URL of the orchestration instance.

The data type of all fields is `string`.

Here is an example response payload for an orchestration instance with `abc123` as its ID (formatted for readability):

```
{
  "id": "abc123",
  "purgeHistoryDeleteUri": "http://localhost:7071/runtime/webhooks/durabletask/instances/abc123?code=XXX",
  "sendEventPostUri":
    "http://localhost:7071/runtime/webhooks/durabletask/instances/abc123/raiseEvent/{eventName}?code=XXX",
  "statusQueryGetUri": "http://localhost:7071/runtime/webhooks/durabletask/instances/abc123?code=XXX",
  "terminatePostUri": "http://localhost:7071/runtime/webhooks/durabletask/instances/abc123/terminate?
  reason={text}&code=XXX",
  "suspendPostUri": "http://localhost:7071/runtime/webhooks/durabletask/instances/abc123/suspend?reason=
  {text}&code=XXX",
  "resumePostUri": "http://localhost:7071/runtime/webhooks/durabletask/instances/abc123/resume?reason=
  {text}&code=XXX"
}
```

The HTTP response is intended to be compatible with the *Polling Consumer Pattern*. It also includes the following notable response headers:

- **Location:** The URL of the status endpoint. This URL contains the same value as the `statusQueryGetUri` field.
- **Retry-After:** The number of seconds to wait between polling operations. The default value is `10`.

For more information on the asynchronous HTTP polling pattern, see the [HTTP async operation tracking](#) documentation.

Get instance status

Gets the status of a specified orchestration instance.

Request

For version 1.x of the Functions runtime, the request is formatted as follows (multiple lines are shown for clarity):

```
GET /admin/extensions/DurableTaskExtension/instances/{instanceId}
?taskHub={taskHub}
&connection={connectionName}
&code={systemKey}
&showHistory=[true|false]
&showHistoryOutput=[true|false]
&showInput=[true|false]
&returnInternalServerErrorOnFailure=[true|false]
```

In version 2.x of the Functions runtime, the URL format has all the same parameters but with a slightly different prefix:

```

GET /runtime/webhooks/durabletask/instances/{instanceId}
?taskHub={taskHub}
&connection={connectionName}
&code={systemKey}
&showHistory=[true|false]
&showHistoryOutput=[true|false]
&showInput=[true|false]
&returnInternalServerErrorOnFailure=[true|false]

```

Request parameters for this API include the default set mentioned previously as well as the following unique parameters:

FIELD	PARAMETER TYPE	DESCRIPTION
<code>instanceId</code>	URL	The ID of the orchestration instance.
<code>showInput</code>	Query string	Optional parameter. If set to <code>false</code> , the function input will not be included in the response payload.
<code>showHistory</code>	Query string	Optional parameter. If set to <code>true</code> , the orchestration execution history will be included in the response payload.
<code>showHistoryOutput</code>	Query string	Optional parameter. If set to <code>true</code> , the function outputs will be included in the orchestration execution history.
<code>createdTimeFrom</code>	Query string	Optional parameter. When specified, filters the list of returned instances that were created at or after the given ISO8601 timestamp.
<code>createdTimeTo</code>	Query string	Optional parameter. When specified, filters the list of returned instances that were created at or before the given ISO8601 timestamp.
<code>runtimeStatus</code>	Query string	Optional parameter. When specified, filters the list of returned instances based on their runtime status. To see the list of possible runtime status values, see the Querying instances article.
<code>returnInternalServerErrorOnFailure</code>	Query string	Optional parameter. If set to <code>true</code> , this API will return an HTTP 500 response instead of a 200 if the instance is in a failure state. This parameter is intended for automated status polling scenarios.

Response

Several possible status code values can be returned.

- **HTTP 200 (OK):** The specified instance is in a completed or failed state.
- **HTTP 202 (Accepted):** The specified instance is in progress.

- **HTTP 400 (Bad Request)**: The specified instance failed or was terminated.
- **HTTP 404 (Not Found)**: The specified instance doesn't exist or has not started running.
- **HTTP 500 (Internal Server Error)**: Returned only when the `returnInternalServerOnFailure` is set to `true` and the specified instance failed with an unhandled exception.

The response payload for the **HTTP 200** and **HTTP 202** cases is a JSON object with the following fields:

FIELD	DATA TYPE	DESCRIPTION
<code>runtimestatus</code>	string	The runtime status of the instance. Values include <i>Running</i> , <i>Pending</i> , <i>Failed</i> , <i>Canceled</i> , <i>Terminated</i> , <i>Completed</i> , <i>Suspended</i> .
<code>input</code>	JSON	The JSON data used to initialize the instance. This field is <code>null</code> if the <code>showInput</code> query string parameter is set to <code>false</code> .
<code>customStatus</code>	JSON	The JSON data used for custom orchestration status. This field is <code>null</code> if not set.
<code>output</code>	JSON	The JSON output of the instance. This field is <code>null</code> if the instance is not in a completed state.
<code>createdTime</code>	string	The time at which the instance was created. Uses ISO 8601 extended notation.
<code>lastUpdatedTime</code>	string	The time at which the instance last persisted. Uses ISO 8601 extended notation.
<code>historyEvents</code>	JSON	A JSON array containing the orchestration execution history. This field is <code>null</code> unless the <code>showHistory</code> query string parameter is set to <code>true</code> .

Here is an example response payload including the orchestration execution history and activity outputs (formatted for readability):

```
{
  "createdTime": "2018-02-28T05:18:49Z",
  "historyEvents": [
    {
      "EventType": "ExecutionStarted",
      "FunctionName": "E1_HelloSequence",
      "Timestamp": "2018-02-28T05:18:49.3452372Z"
    },
    {
      "EventType": "TaskCompleted",
      "FunctionName": "E1_SayHello",
      "Result": "Hello Tokyo!",
      "ScheduledTime": "2018-02-28T05:18:51.3939873Z",
      "Timestamp": "2018-02-28T05:18:52.2895622Z"
    },
    {
      "EventType": "TaskCompleted",
      "FunctionName": "E1_SayHello",
      "Result": "Hello Seattle!",
      "ScheduledTime": "2018-02-28T05:18:52.8755705Z",
      "Timestamp": "2018-02-28T05:18:53.1765771Z"
    },
    {
      "EventType": "TaskCompleted",
      "FunctionName": "E1_SayHello",
      "Result": "Hello London!",
      "ScheduledTime": "2018-02-28T05:18:53.5170791Z",
      "Timestamp": "2018-02-28T05:18:53.891081Z"
    },
    {
      "EventType": "ExecutionCompleted",
      "OrchestrationStatus": "Completed",
      "Result": [
        "Hello Tokyo!",
        "Hello Seattle!",
        "Hello London!"
      ],
      "Timestamp": "2018-02-28T05:18:54.3660895Z"
    }
  ],
  "input": null,
  "customStatus": { "nextActions": [ "A", "B", "C" ], "foo": 2 },
  "lastUpdatedTime": "2018-02-28T05:18:54Z",
  "output": [
    "Hello Tokyo!",
    "Hello Seattle!",
    "Hello London!"
  ],
  "runtimeStatus": "Completed"
}
```

The HTTP 202 response also includes a **Location** response header that references the same URL as the `statusQueryGetUri` field mentioned previously.

Get all instances status

You can also query the status of all instances by removing the `instanceId` from the 'Get instance status' request. In this case, the basic parameters are the same as the 'Get instance status'. Query string parameters for filtering are also supported.

Request

For version 1.x of the Functions runtime, the request is formatted as follows (multiple lines are shown for clarity):

```

GET /admin/extensions/DurableTaskExtension/instances
?taskHub={taskHub}
&connection={connectionName}
&code={systemKey}
&createdTimeFrom={timestamp}
&createdTimeTo={timestamp}
&runtimeStatus={runtimeStatus1,runtimeStatus2,...}
&instanceIdPrefix={prefix}
&showInput=[true|false]
&top={integer}

```

In version 2.x of the Functions runtime, the URL format has all the same parameters but with a slightly different prefix:

```

GET /runtime/webhooks/durableTask/instances?
taskHub={taskHub}
&connection={connectionName}
&code={systemKey}
&createdTimeFrom={timestamp}
&createdTimeTo={timestamp}
&runtimeStatus={runtimeStatus1,runtimeStatus2,...}
&instanceIdPrefix={prefix}
&showInput=[true|false]
&top={integer}

```

Request parameters for this API include the default set mentioned previously as well as the following unique parameters:

FIELD	PARAMETER TYPE	DESCRIPTION
<code>showInput</code>	Query string	Optional parameter. If set to <code>false</code> , the function input will not be included in the response payload.
<code>showHistoryOutput</code>	Query string	Optional parameter. If set to <code>true</code> , the function outputs will be included in the orchestration execution history.
<code>createdTimeFrom</code>	Query string	Optional parameter. When specified, filters the list of returned instances that were created at or after the given ISO8601 timestamp.
<code>createdTimeTo</code>	Query string	Optional parameter. When specified, filters the list of returned instances that were created at or before the given ISO8601 timestamp.
<code>runtimeStatus</code>	Query string	Optional parameter. When specified, filters the list of returned instances based on their runtime status. To see the list of possible runtime status values, see the Querying instances article.

FIELD	PARAMETER TYPE	DESCRIPTION
<code>instanceIdPrefix</code>	Query string	Optional parameter. When specified, filters the list of returned instances to include only instances whose instance ID starts with the specified prefix string. Available starting with version 2.7.2 of the extension.
<code>top</code>	Query string	Optional parameter. When specified, limits the number of instances returned by the query.

Response

Here is an example of response payloads including the orchestration status (formatted for readability):

```
[
  {
    "instanceId": "7af46ff000564c65aafbfe99d07c32a5",
    "runtimeStatus": "Completed",
    "input": null,
    "customStatus": null,
    "output": [
      "Hello Tokyo!",
      "Hello Seattle!",
      "Hello London!"
    ],
    "createdTime": "2018-06-04T10:46:39Z",
    "lastUpdatedTime": "2018-06-04T10:46:47Z"
  },
  {
    "instanceId": "80eb7dd5c22f4eeba9f42b062794321e",
    "runtimeStatus": "Running",
    "input": null,
    "customStatus": null,
    "output": null,
    "createdTime": "2018-06-04T15:18:28Z",
    "lastUpdatedTime": "2018-06-04T15:18:38Z"
  },
  {
    "instanceId": "9124518926db408ab8dfe84822aba2b1",
    "runtimeStatus": "Completed",
    "input": null,
    "customStatus": null,
    "output": [
      "Hello Tokyo!",
      "Hello Seattle!",
      "Hello London!"
    ],
    "createdTime": "2018-06-04T10:46:54Z",
    "lastUpdatedTime": "2018-06-04T10:47:03Z"
  },
  {
    "instanceId": "d100b90b903c4009ba1a90868331b11b",
    "runtimeStatus": "Pending",
    "input": null,
    "customStatus": null,
    "output": null,
    "createdTime": "2018-06-04T15:18:39Z",
    "lastUpdatedTime": "2018-06-04T15:18:39Z"
  }
]
```

NOTE

This operation can be very expensive in terms of Azure Storage I/O if you are using the [default Azure Storage provider](#) and if there are a lot of rows in the Instances table. More details on Instance table can be found in the [Azure Storage provider](#) documentation.

If more results exist, a continuation token is returned in the response header. The name of the header is

`x-ms-continuation-token`.

Caution

The query result may return fewer items than the limit specified by `top`. When receiving results, you should therefore *always* check to see if there is a continuation token.

If you set continuation token value in the next request header, you can get the next page of results. This name of the request header is also `x-ms-continuation-token`.

Purge single instance history

Deletes the history and related artifacts for a specified orchestration instance.

Request

For version 1.x of the Functions runtime, the request is formatted as follows (multiple lines are shown for clarity):

```
DELETE /admin/extensions/DurableTaskExtension/instances/{instanceId}
?taskHub={taskHub}
&connection={connection}
&code={systemKey}
```

In version 2.x of the Functions runtime, the URL format has all the same parameters but with a slightly different prefix:

```
DELETE /runtime/webhooks/durabletask/instances/{instanceId}
?taskHub={taskHub}
&connection={connection}
&code={systemKey}
```

Request parameters for this API include the default set mentioned previously as well as the following unique parameters:

FIELD	PARAMETER TYPE	DESCRIPTION
<code>instanceId</code>	URL	The ID of the orchestration instance.

Response

The following HTTP status code values can be returned.

- **HTTP 200 (OK)**: The instance history has been purged successfully.
- **HTTP 404 (Not Found)**: The specified instance doesn't exist.

The response payload for the **HTTP 200** case is a JSON object with the following field:

FIELD	DATA TYPE	DESCRIPTION
<code>instancesDeleted</code>	integer	The number of instances deleted. For the single instance case, this value should always be <code>1</code> .

Here is an example response payload (formatted for readability):

```
{
  "instancesDeleted": 1
}
```

Purge multiple instance histories

You can also delete the history and related artifacts for multiple instances within a task hub by removing the `{instanceId}` from the 'Purge single instance history' request. To selectively purge instance history, use the same filters described in the 'Get all instances status' request.

Request

For version 1.x of the Functions runtime, the request is formatted as follows (multiple lines are shown for clarity):

```
DELETE /admin/extensions/DurableTaskExtension/instances
?taskHub={taskHub}
&connection={connectionName}
&code={systemKey}
&createdTimeFrom={timestamp}
&createdTimeTo={timestamp}
&runtimeStatus={runtimeStatus1,runtimeStatus2,...}
```

In version 2.x of the Functions runtime, the URL format has all the same parameters but with a slightly different prefix:

```
DELETE /runtime/webhooks/durabletask/instances
?taskHub={taskHub}
&connection={connectionName}
&code={systemKey}
&createdTimeFrom={timestamp}
&createdTimeTo={timestamp}
&runtimeStatus={runtimeStatus1,runtimeStatus2,...}
```

Request parameters for this API include the default set mentioned previously as well as the following unique parameters:

FIELD	PARAMETER TYPE	DESCRIPTION
<code>createdTimeFrom</code>	Query string	Filters the list of purged instances that were created at or after the given ISO8601 timestamp.
<code>createdTimeTo</code>	Query string	Optional parameter. When specified, filters the list of purged instances that were created at or before the given ISO8601 timestamp.

FIELD	PARAMETER TYPE	DESCRIPTION
runtimeStatus	Query string	Optional parameter. When specified, filters the list of purged instances based on their runtime status. To see the list of possible runtime status values, see the Querying instances article.

NOTE

This operation can be very expensive in terms of Azure Storage I/O if you are using the [default Azure Storage provider](#) and if there are many rows in the Instances and/or History tables. More details on these tables can be found in the [Performance and scale in Durable Functions \(Azure Functions\)](#) documentation.

Response

The following HTTP status code values can be returned.

- **HTTP 200 (OK):** The instance history has been purged successfully.
- **HTTP 404 (Not Found):** No instances were found that match the filter expression.

The response payload for the **HTTP 200** case is a JSON object with the following field:

FIELD	DATA TYPE	DESCRIPTION
instancesDeleted	integer	The number of instances deleted.

Here is an example response payload (formatted for readability):

```
{
    "instancesDeleted": 250
}
```

Raise event

Sends an event notification message to a running orchestration instance.

Request

For version 1.x of the Functions runtime, the request is formatted as follows (multiple lines are shown for clarity):

```
POST /admin/extensions/DurableTaskExtension/instances/{instanceId}/raiseEvent/{eventName}
?taskHub={taskHub}
&connection={connectionName}
&code={systemKey}
```

In version 2.x of the Functions runtime, the URL format has all the same parameters but with a slightly different prefix:

```
POST /runtime/webhooks/durabletask/instances/{instanceId}/raiseEvent/{eventName}
?taskHub={taskHub}
&connection={connectionName}
&code={systemKey}
```

Request parameters for this API include the default set mentioned previously as well as the following unique parameters:

FIELD	PARAMETER TYPE	DESCRIPTION
<code>instanceId</code>	URL	The ID of the orchestration instance.
<code>eventName</code>	URL	The name of the event that the target orchestration instance is waiting on.
<code>{content}</code>	Request content	The JSON-formatted event payload.

Response

Several possible status code values can be returned.

- **HTTP 202 (Accepted)**: The raised event was accepted for processing.
- **HTTP 400 (Bad request)**: The request content was not of type `application/json` or was not valid JSON.
- **HTTP 404 (Not Found)**: The specified instance was not found.
- **HTTP 410 (Gone)**: The specified instance has completed or failed and cannot process any raised events.

Here is an example request that sends the JSON string `"incr"` to an instance waiting for an event named `operation`:

```
POST /admin/extensions/DurableTaskExtension/instances/bcf6fb5067b046fbb021b52ba7deae5a/raiseEvent/operation?  
taskHub=DurableFunctionsHub&connection=Storage&code=XXX  
Content-Type: application/json  
Content-Length: 6  
  
"incr"
```

The responses for this API do not contain any content.

Terminate instance

Terminates a running orchestration instance.

Request

For version 1.x of the Functions runtime, the request is formatted as follows (multiple lines are shown for clarity):

```
POST /admin/extensions/DurableTaskExtension/instances/{instanceId}/terminate  
?taskHub={taskHub}  
&connection={connectionName}  
&code={systemKey}  
&reason={text}
```

In version 2.x of the Functions runtime, the URL format has all the same parameters but with a slightly different prefix:

```
POST /runtime/webhooks/durabletask/instances/{instanceId}/terminate  
?taskHub={taskHub}  
&connection={connectionName}  
&code={systemKey}  
&reason={text}
```

Request parameters for this API include the default set mentioned previously as well as the following unique parameter.

FIELD	PARAMETER TYPE	DESCRIPTION
instanceId	URL	The ID of the orchestration instance.
reason	Query string	Optional. The reason for terminating the orchestration instance.

Response

Several possible status code values can be returned.

- **HTTP 202 (Accepted)**: The terminate request was accepted for processing.
- **HTTP 404 (Not Found)**: The specified instance was not found.
- **HTTP 410 (Gone)**: The specified instance has completed or failed.

Here is an example request that terminates a running instance and specifies a reason of **buggy**:

```
POST /admin/extensions/DurableTaskExtension/instances/bcf6fb5067b046fbb021b52ba7deae5a/terminate?  
reason=buggy&taskHub=DurableFunctionsHub&connection=Storage&code=XXX
```

The responses for this API do not contain any content.

Suspend instance (preview)

Suspends a running orchestration instance.

Request

In version 2.x of the Functions runtime, the request is formatted as follows (multiple lines are shown for clarity):

```
POST /runtime/webhooks/durabletask/instances/{instanceId}/suspend  
?reason={text}  
&taskHub={taskHub}  
&connection={connectionName}  
&code={systemKey}
```

FIELD	PARAMETER TYPE	DESCRIPTION
instanceId	URL	The ID of the orchestration instance.
reason	Query string	Optional. The reason for suspending the orchestration instance.

Several possible status code values can be returned.

- **HTTP 202 (Accepted)**: The suspend request was accepted for processing.
- **HTTP 404 (Not Found)**: The specified instance was not found.
- **HTTP 410 (Gone)**: The specified instance has completed, failed, or terminated.

The responses for this API do not contain any content.

Resume instance (preview)

Resumes a suspended orchestration instance.

Request

In version 2.x of the Functions runtime, the request is formatted as follows (multiple lines are shown for clarity):

```
POST /runtime/webhooks/durabletask/instances/{instanceId}/resume  
?reason={text}  
&taskHub={taskHub}  
&connection={connectionName}  
&code={systemKey}
```

FIELD	PARAMETER TYPE	DESCRIPTION
instanceId	URL	The ID of the orchestration instance.
reason	Query string	Optional. The reason for resuming the orchestration instance.

Several possible status code values can be returned.

- **HTTP 202 (Accepted)**: The resume request was accepted for processing.
- **HTTP 404 (Not Found)**: The specified instance was not found.
- **HTTP 410 (Gone)**: The specified instance has completed, failed, or terminated.

The responses for this API do not contain any content.

Rewind instance (preview)

Restores a failed orchestration instance into a running state by replaying the most recent failed operations.

Request

For version 1.x of the Functions runtime, the request is formatted as follows (multiple lines are shown for clarity):

```
POST /admin/extensions/DurableTaskExtension/instances/{instanceId}/rewind  
?taskHub={taskHub}  
&connection={connectionName}  
&code={systemKey}  
&reason={text}
```

In version 2.x of the Functions runtime, the URL format has all the same parameters but with a slightly different prefix:

```
POST /runtime/webhooks/durabletask/instances/{instanceId}/rewind  
?taskHub={taskHub}  
&connection={connectionName}  
&code={systemKey}  
&reason={text}
```

Request parameters for this API include the default set mentioned previously as well as the following unique parameter.

FIELD	PARAMETER TYPE	DESCRIPTION
<code>instanceId</code>	URL	The ID of the orchestration instance.
<code>reason</code>	Query string	Optional. The reason for rewinding the orchestration instance.

Response

Several possible status code values can be returned.

- **HTTP 202 (Accepted)**: The rewind request was accepted for processing.
- **HTTP 404 (Not Found)**: The specified instance was not found.
- **HTTP 410 (Gone)**: The specified instance has completed or was terminated.

Here is an example request that rewinds a failed instance and specifies a reason of **fixed**:

```
POST /admin/extensions/DurableTaskExtension/instances/bcf6fb5067b046fbb021b52ba7deae5a/rewind?
reason=fixed&taskHub=DurableFunctionsHub&connection=Storage&code=XXX
```

The responses for this API do not contain any content.

Signal entity

Sends a one-way operation message to a [Durable Entity](#). If the entity doesn't exist, it will be created automatically.

NOTE

Durable entities are available starting in Durable Functions 2.0.

Request

The HTTP request is formatted as follows (multiple lines are shown for clarity):

```
POST /runtime/webhooks/durabletask/entities/{entityName}/{entityKey}
?taskHub={taskHub}
&connection={connectionName}
&code={systemKey}
&op={operationName}
```

Request parameters for this API include the default set mentioned previously as well as the following unique parameters:

FIELD	PARAMETER TYPE	DESCRIPTION
<code>entityName</code>	URL	The name (type) of the entity.
<code>entityKey</code>	URL	The key (unique ID) of the entity.
<code>op</code>	Query string	Optional. The name of the user-defined operation to invoke.
<code>{content}</code>	Request content	The JSON-formatted event payload.

Here is an example request that sends a user-defined "Add" message to a `Counter` entity named `steps`. The content of the message is the value `5`. If the entity does not already exist, it will be created by this request:

```
POST /runtime/webhooks/durabletask/entities/Counter/steps?op=Add
Content-Type: application/json

5
```

NOTE

By default with [class-based entities in .NET](#), specifying the `op` value of `delete` will delete the state of an entity. If the entity defines an operation named `delete`, however, that user-defined operation will be invoked instead.

Response

This operation has several possible responses:

- **HTTP 202 (Accepted)**: The signal operation was accepted for asynchronous processing.
- **HTTP 400 (Bad request)**: The request content was not of type `application/json`, was not valid JSON, or had an invalid `entityKey` value.
- **HTTP 404 (Not Found)**: The specified `entityName` was not found.

A successful HTTP request does not contain any content in the response. A failed HTTP request may contain JSON-formatted error information in the response content.

Get entity

Gets the state of the specified entity.

Request

The HTTP request is formatted as follows (multiple lines are shown for clarity):

```
GET /runtime/webhooks/durabletask/entities/{entityName}/{entityKey}
?taskHub={taskHub}
&connection={connectionName}
&code={systemKey}
```

Response

This operation has two possible responses:

- **HTTP 200 (OK)**: The specified entity exists.
- **HTTP 404 (Not Found)**: The specified entity was not found.

A successful response contains the JSON-serialized state of the entity as its content.

Example

The following example HTTP request gets the state of an existing `Counter` entity named `steps`:

```
GET /runtime/webhooks/durabletask/entities/Counter/steps
```

If the `Counter` entity simply contained a number of steps saved in a `currentValue` field, the response content might look like the following (formatted for readability):

```
{  
    "currentValue": 5  
}
```

List entities

You can query for multiple entities by the entity name or by the last operation date.

Request

The HTTP request is formatted as follows (multiple lines are shown for clarity):

```
GET /runtime/webhooks/durabletask/entities/{entityName}  
?taskHub={taskHub}  
&connection={connectionName}  
&code={systemKey}  
&lastOperationTimeFrom={timestamp}  
&lastOperationTimeTo={timestamp}  
&fetchState=[true|false]  
&top={integer}
```

Request parameters for this API include the default set mentioned previously as well as the following unique parameters:

FIELD	PARAMETER TYPE	DESCRIPTION
<code>entityName</code>	URL	Optional. When specified, filters the list of returned entities by their entity name (case-insensitive).
<code>fetchState</code>	Query string	Optional parameter. If set to <code>true</code> , the entity state will be included in the response payload.
<code>lastOperationTimeFrom</code>	Query string	Optional parameter. When specified, filters the list of returned entities that processed operations after the given ISO8601 timestamp.
<code>lastOperationTimeTo</code>	Query string	Optional parameter. When specified, filters the list of returned entities that processed operations before the given ISO8601 timestamp.
<code>top</code>	Query string	Optional parameter. When specified, limits the number of entities returned by the query.

Response

A successful HTTP 200 response contains a JSON-serialized array of entities and optionally the state of each entity.

By default the operation returns the first 100 entities that match the query criteria. The caller can specify a query string parameter value for `top` to return a different maximum number of results. If more results exist beyond what is returned, a continuation token is also returned in the response header. The name of the header is

`x-ms-continuation-token`.

If you set continuation token value in the next request header, you can get the next page of results. This name of the request header is also `x-ms-continuation-token`.

Example - list all entities

The following example HTTP request lists all entities in the task hub:

```
GET /runtime/webhooks/durabletask/entities
```

The response JSON may look like the following (formatted for readability):

```
[  
  {  
    "entityId": { "key": "cats", "name": "counter" },  
    "lastOperationTime": "2019-12-18T21:45:44.6326361Z",  
  },  
  {  
    "entityId": { "key": "dogs", "name": "counter" },  
    "lastOperationTime": "2019-12-18T21:46:01.9477382Z"  
  },  
  {  
    "entityId": { "key": "mice", "name": "counter" },  
    "lastOperationTime": "2019-12-18T21:46:15.4626159Z"  
  },  
  {  
    "entityId": { "key": "radio", "name": "device" },  
    "lastOperationTime": "2019-12-18T21:46:18.2616154Z"  
  },  
]
```

Example - filtering the list of entities

The following example HTTP request lists just the first two entities of type `counter` and also fetches their state:

```
GET /runtime/webhooks/durabletask/entities/counter?top=2&fetchState=true
```

The response JSON may look like the following (formatted for readability):

```
[  
  {  
    "entityId": { "key": "cats", "name": "counter" },  
    "lastOperationTime": "2019-12-18T21:45:44.6326361Z",  
    "state": { "value": 9 }  
  },  
  {  
    "entityId": { "key": "dogs", "name": "counter" },  
    "lastOperationTime": "2019-12-18T21:46:01.9477382Z",  
    "state": { "value": 10 }  
  }  
]
```

Next steps

[Learn how to use Application Insights to monitor your durable functions](#)

Azure Storage provider (Azure Functions)

10/5/2022 • 18 minutes to read • [Edit Online](#)

This document describes the characteristics of the Durable Functions Azure Storage provider, with a focus on performance and scalability aspects. The Azure Storage provider is the default provider. It stores instance states and queues in an Azure Storage (classic) account.

NOTE

For more information on the supported storage providers for Durable Functions and how they compare, see the [Durable Functions storage providers](#) documentation.

In the Azure Storage provider, all function execution is driven by Azure Storage queues. Orchestration and entity status and history are stored in Azure Tables. Azure Blobs and blob leases are used to distribute orchestration instances and entities across multiple app instances (also known as *workers* or simply *VMs*). This section goes into more detail on the various Azure Storage artifacts and how they impact performance and scalability.

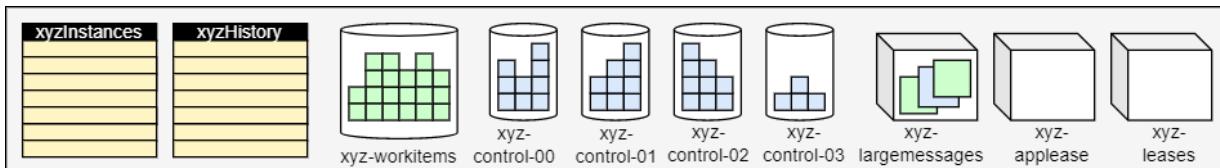
Storage representation

A [task hub](#) durably persists all instance states and all messages. For a quick overview of how these are used to track the progress of an orchestration, see the [task hub execution example](#).

The Azure Storage provider represents the task hub in storage using the following components:

- Two Azure Tables store the instance states.
- One Azure Queue stores the activity messages.
- One or more Azure Queues store the instance messages. Each of these so-called *control queues* represents a [partition](#) that is assigned a subset of all instance messages, based on the hash of the instance ID.
- A few extra blob containers used for lease blobs and/or large messages.

For example, a task hub named `xyz` with `PartitionCount = 4` contains the following queues and tables:



Next, we describe these components and the role they play in more detail.

History table

The **History** table is an Azure Storage table that contains the history events for all orchestration instances within a task hub. The name of this table is in the form `TaskHubNameHistory`. As instances run, new rows are added to this table. The partition key of this table is derived from the instance ID of the orchestration. Instance IDs are random by default, ensuring optimal distribution of internal partitions in Azure Storage. The row key for this table is a sequence number used for ordering the history events.

When an orchestration instance needs to run, the corresponding rows of the History table are loaded into memory using a range query within a single table partition. These *history events* are then replayed into the orchestrator function code to get it back into its previously checkpointed state. The use of execution history to rebuild state in this way is influenced by the [Event Sourcing pattern](#).

TIP

Orchestration data stored in the History table includes output payloads from activity and sub-orchestrator functions. Payloads from external events are also stored in the History table. Because the full history is loaded into memory every time an orchestrator needs to execute, a large enough history can result in significant memory pressure on a given VM. The length and size of the orchestration history can be reduced by splitting large orchestrations into multiple sub-orchestrations or by reducing the size of outputs returned by the activity and sub-orchestrator functions it calls. Alternatively, you can reduce memory usage by lowering per-VM [concurrency throttles](#) to limit how many orchestrations are loaded into memory concurrently.

Instances table

The **Instances** table contains the statuses of all orchestration and entity instances within a task hub. As instances are created, new rows are added to this table. The partition key of this table is the orchestration instance ID or entity key and the row key is an empty string. There is one row per orchestration or entity instance.

This table is used to satisfy [instance query requests from code](#) as well as [status query HTTP API](#) calls. It is kept eventually consistent with the contents of the **History** table mentioned previously. The use of a separate Azure Storage table to efficiently satisfy instance query operations in this way is influenced by the [Command and Query Responsibility Segregation \(CQRS\) pattern](#).

TIP

The partitioning of the *Instances* table allows it to store millions of orchestration instances without any noticeable impact on runtime performance or scale. However, the number of instances can have a significant impact on [multi-instance query](#) performance. To control the amount of data stored in these tables, consider periodically [purging old instance data](#).

Queues

Orchestrator, entity, and activity functions are all triggered by internal queues in the function app's task hub. Using queues in this way provides reliable "at-least-once" message delivery guarantees. There are two types of queues in Durable Functions: the **control queue** and the **work-item queue**.

The work-item queue

There is one work-item queue per task hub in Durable Functions. It's a basic queue and behaves similarly to any other `queueTrigger` queue in Azure Functions. This queue is used to trigger stateless *activity functions* by dequeuing a single message at a time. Each of these messages contains activity function inputs and additional metadata, such as which function to execute. When a Durable Functions application scales out to multiple VMs, these VMs all compete to acquire tasks from the work-item queue.

Control queue(s)

There are multiple *control queues* per task hub in Durable Functions. A *control queue* is more sophisticated than the simpler work-item queue. Control queues are used to trigger the stateful orchestrator and entity functions. Because the orchestrator and entity function instances are stateful singletons, it's important that each orchestration or entity is only processed by one worker at a time. To achieve this constraint, each orchestration instance or entity is assigned to a single control queue. These control queues are load balanced across workers to ensure that each queue is only processed by one worker at a time. More details on this behavior can be found in subsequent sections.

Control queues contain a variety of orchestration lifecycle message types. Examples include [orchestrator control messages](#), activity function *response* messages, and timer messages. As many as 32 messages will be dequeued from a control queue in a single poll. These messages contain payload data as well as metadata including which orchestration instance it is intended for. If multiple dequeued messages are intended for the same orchestration instance, they will be processed as a batch.

Control queue messages are constantly polled using a background thread. The batch size of each queue poll is controlled by the `controlQueueBatchSize` setting in `host.json` and has a default of 32 (the maximum value supported by Azure Queues). The maximum number of prefetched control-queue messages that are buffered in memory is controlled by the `controlQueueBufferThreshold` setting in `host.json`. The default value for `controlQueueBufferThreshold` varies depending on a variety of factors, including the type of hosting plan. For more information on these settings, see the [host.json schema](#) documentation.

TIP

Increasing the value for `controlQueueBufferThreshold` allows a single orchestration or entity to process events faster. However, increasing this value can also result in higher memory usage. The higher memory usage is partly due to pulling more messages off the queue and partly due to fetching more orchestration histories into memory. Reducing the value for `controlQueueBufferThreshold` can therefore be an effective way to reduce memory usage.

Queue polling

The durable task extension implements a random exponential back-off algorithm to reduce the effect of idle-queue polling on storage transaction costs. When a message is found, the runtime immediately checks for another message. When no message is found, it waits for a period of time before trying again. After subsequent failed attempts to get a queue message, the wait time continues to increase until it reaches the maximum wait time, which defaults to 30 seconds.

The maximum polling delay is configurable via the `maxQueuePollingInterval` property in the [host.json file](#). Setting this property to a higher value could result in higher message processing latencies. Higher latencies would be expected only after periods of inactivity. Setting this property to a lower value could result in [higher storage costs](#) due to increased storage transactions.

NOTE

When running in the Azure Functions Consumption and Premium plans, the [Azure Functions Scale Controller](#) will poll each control and work-item queue once every 10 seconds. This additional polling is necessary to determine when to activate function app instances and to make scale decisions. At the time of writing, this 10 second interval is constant and cannot be configured.

Orchestration start delays

Orchestrations instances are started by putting an `ExecutionStarted` message in one of the task hub's control queues. Under certain conditions, you may observe multi-second delays between when an orchestration is scheduled to run and when it actually starts running. During this time interval, the orchestration instance remains in the `Pending` state. There are two potential causes of this delay:

- **Backlogged control queues:** If the control queue for this instance contains a large number of messages, it may take time before the `ExecutionStarted` message is received and processed by the runtime. Message backlogs can happen when orchestrations are processing lots of events concurrently. Events that go into the control queue include orchestration start events, activity completions, durable timers, termination, and external events. If this delay happens under normal circumstances, consider creating a new task hub with a larger number of partitions. Configuring more partitions will cause the runtime to create more control queues for load distribution. Each partition corresponds to 1:1 with a control queue, with a maximum of 16 partitions.
- **Back off polling delays:** Another common cause of orchestration delays is the [previously described back-off polling behavior for control queues](#). However, this delay is only expected when an app is scaled out to two or more instances. If there is only one app instance or if the app instance that starts the orchestration is also the same instance that is polling the target control queue, then there will not be a queue polling delay. Back off polling delays can be reduced by updating the `host.json` settings, as

described previously.

Blobs

In most cases, Durable Functions doesn't use Azure Storage Blobs to persist data. However, queues and tables have [size limits](#) that can prevent Durable Functions from persisting all of the required data into a storage row or queue message. For example, when a piece of data that needs to be persisted to a queue is greater than 45 KB when serialized, Durable Functions will compress the data and store it in a blob instead. When persisting data to blob storage in this way, Durable Function stores a reference to that blob in the table row or queue message. When Durable Functions needs to retrieve the data it will automatically fetch it from the blob. These blobs are stored in the blob container `<taskhub>-largemessages`.

Performance considerations

The extra compression and blob operation steps for large messages can be expensive in terms of CPU and I/O latency costs. Additionally, Durable Functions needs to load persisted data in memory, and may do so for many different function executions at the same time. As a result, persisting large data payloads can cause high memory usage as well. To minimize memory overhead, consider persisting large data payloads manually (for example, in blob storage) and instead pass around references to this data. This way your code can load the data only when needed to avoid redundant loads during [orchestrator function replays](#). However, storing payloads to local disks is *not* recommended since on-disk state is not guaranteed to be available since functions may execute on different VMs throughout their lifetimes.

Storage account selection

The queues, tables, and blobs used by Durable Functions are created in a configured Azure Storage account. The account to use can be specified using the `durableTask/storageProvider/connectionStringName` setting (or `durableTask/azureStorageConnectionStringName` setting in Durable Functions 1.x) in the `host.json` file.

Durable Functions 2.x

```
{  
  "extensions": {  
    "durableTask": {  
      "storageProvider": {  
        "connectionStringName": "MyStorageAccountAppSetting"  
      }  
    }  
  }  
}
```

Durable Functions 1.x

```
{  
  "extensions": {  
    "durableTask": {  
      "azureStorageConnectionStringName": "MyStorageAccountAppSetting"  
    }  
  }  
}
```

If not specified, the default `AzureWebJobsStorage` storage account is used. For performance-sensitive workloads, however, configuring a non-default storage account is recommended. Durable Functions uses Azure Storage heavily, and using a dedicated storage account isolates Durable Functions storage usage from the internal usage by the Azure Functions host.

NOTE

Standard general purpose Azure Storage accounts are required when using the Azure Storage provider. All other storage account types are not supported. We highly recommend using legacy v1 general purpose storage accounts for Durable Functions. The newer v2 storage accounts can be significantly more expensive for Durable Functions workloads. For more information on Azure Storage account types, see the [Storage account overview](#) documentation.

Orchestrator scale-out

While activity functions can be scaled out infinitely by adding more VMs elastically, individual orchestrator instances and entities are constrained to inhabit a single partition and the maximum number of partitions is bounded by the `partitionCount` setting in your `host.json`.

NOTE

Generally speaking, orchestrator functions are intended to be lightweight and should not require large amounts of computing power. It is therefore not necessary to create a large number of control-queue partitions to get great throughput for orchestrations. Most of the heavy work should be done in stateless activity functions, which can be scaled out infinitely.

The number of control queues is defined in the `host.json` file. The following example `host.json` snippet sets the `durableTask/storageProvider/partitionCount` property (or `durableTask/partitionCount` in Durable Functions 1.x) to `3`. Note that there are as many control queues as there are partitions.

Durable Functions 2.x

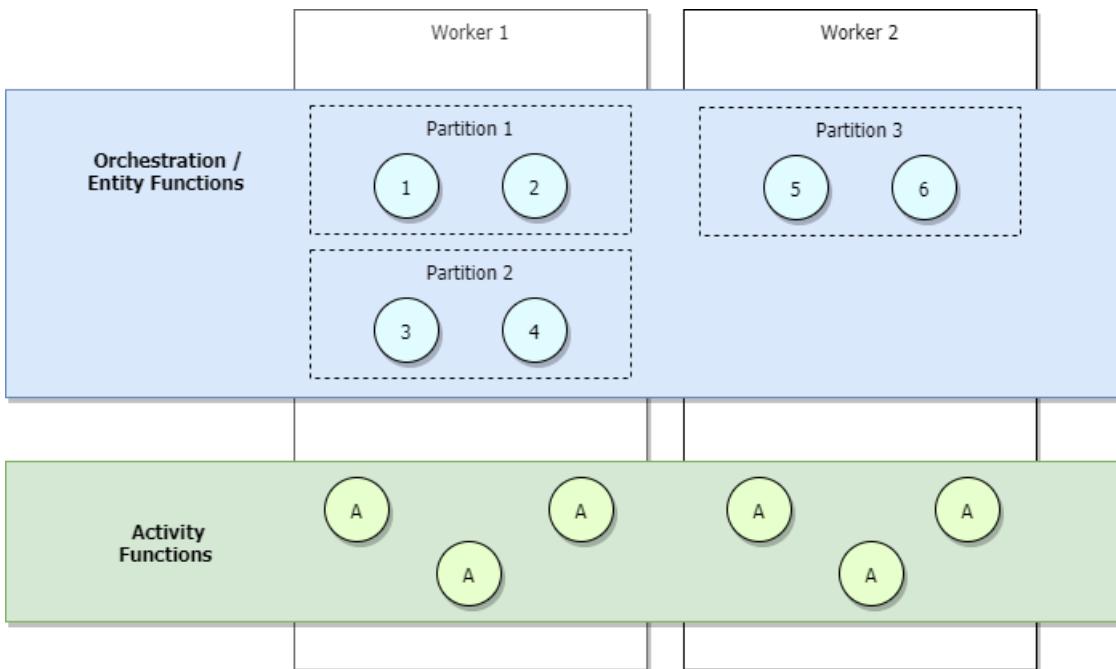
```
{  
  "extensions": {  
    "durableTask": {  
      "storageProvider": {  
        "partitionCount": 3  
      }  
    }  
  }  
}
```

Durable Functions 1.x

```
{  
  "extensions": {  
    "durableTask": {  
      "partitionCount": 3  
    }  
  }  
}
```

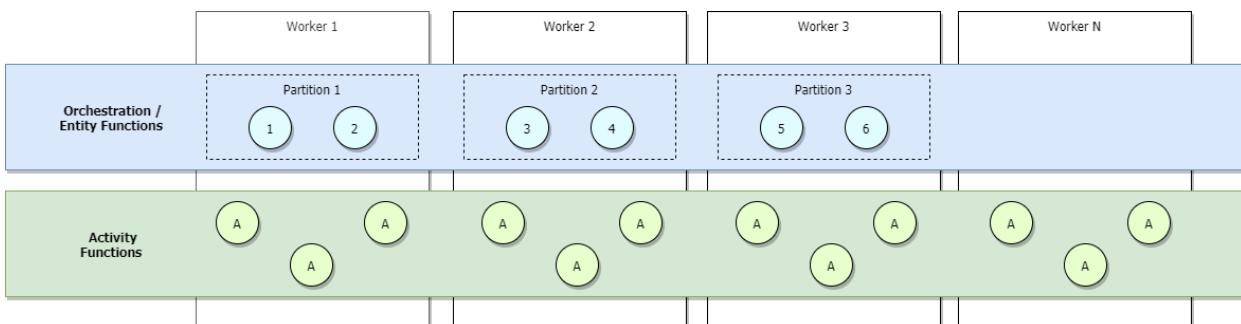
A task hub can be configured with between 1 and 16 partitions. If not specified, the default partition count is 4.

During low traffic scenarios, your application will be scaled-in, so partitions will be managed by a small number of workers. As an example, consider the diagram below.



In the previous diagram, we see that orchestrators 1 through 6 are load balanced across partitions. Similarly, partitions, like activities, are load balanced across workers. Partitions are load-balanced across workers regardless of the number of orchestrators that get started.

If you're running on the Azure Functions Consumption or Elastic Premium plans, or if you have load-based auto-scaling configured, more workers will get allocated as traffic increases and partitions will eventually load balance across all workers. If we continue to scale out, eventually each partition will eventually be managed by a single worker. Activities, on the other hand, will continue to be load-balanced across all workers. This is shown in the image below.



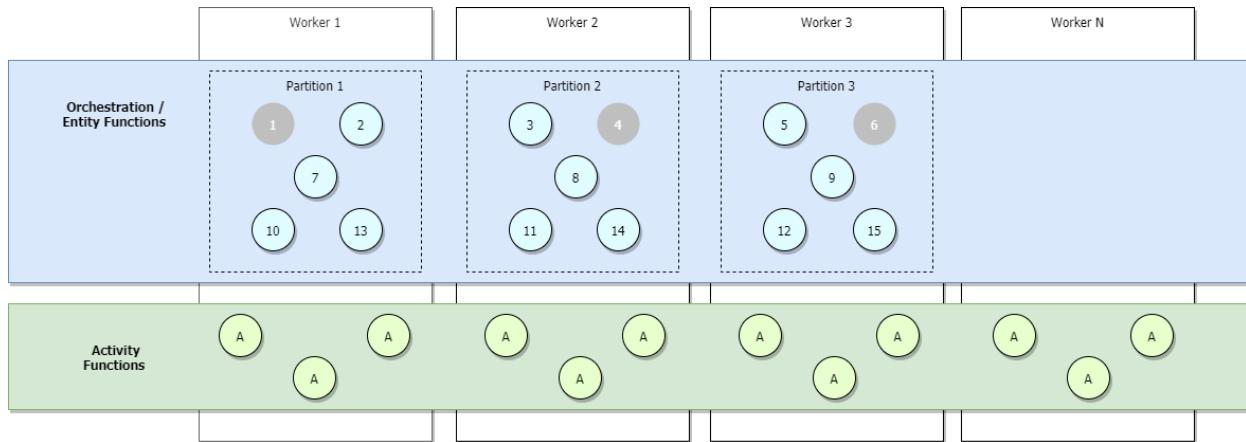
The upper-bound of the maximum number of concurrent *active* orchestrations at *any given time* is equal to the number of workers allocated to your application *times* your value for `maxConcurrentOrchestratorFunctions`. This upper-bound can be made more precise when your partitions are fully scaled-out across workers. When fully scaled-out, and since each worker will have only a single Functions host instance, the maximum number of *active* concurrent orchestrator instances will be equal to your number of partitions *times* your value for `maxConcurrentOrchestratorFunctions`.

NOTE

In this context, *active* means that an orchestration or entity is loaded into memory and processing *new events*. If the orchestration or entity is waiting for more events, such as the return value of an activity function, it gets unloaded from memory and is no longer considered *active*. Orchestrations and entities will be subsequently reloaded into memory only when there are new events to process. There's no practical maximum number of *total* orchestrations or entities that can run on a single VM, even if they're all in the "Running" state. The only limitation is the number of *concurrently active* orchestration or entity instances.

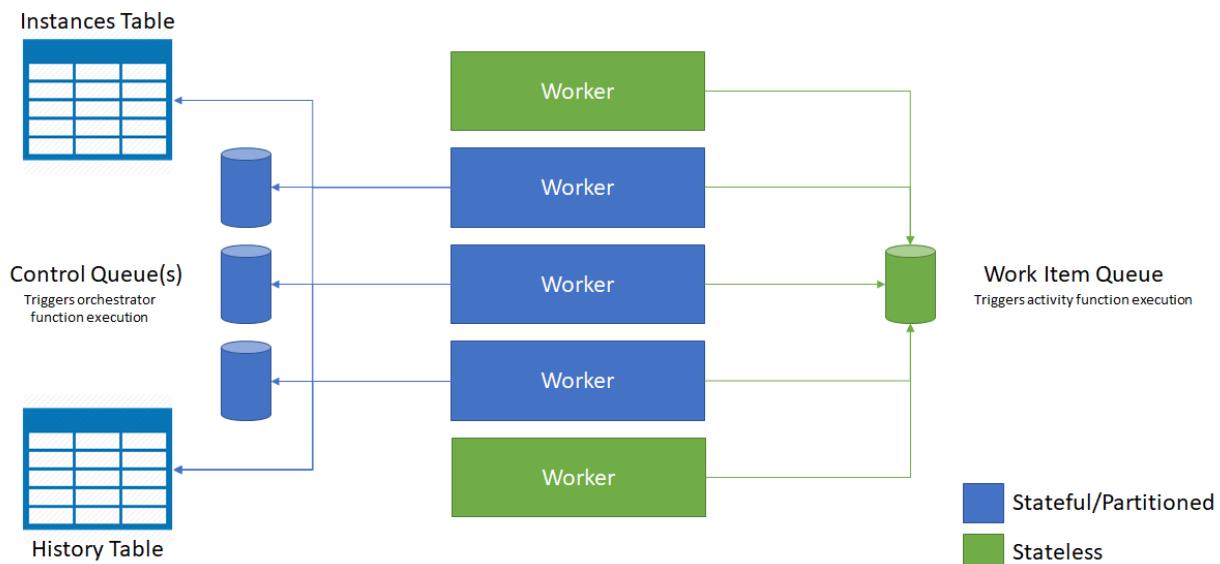
The image below illustrates a fully scaled-out scenario where more orchestrators are added but some are

inactive, shown in grey.



During scale-out, control queue leases may be redistributed across Functions host instances to ensure that partitions are evenly distributed. These leases are internally implemented as Azure Blob storage leases and ensure that any individual orchestration instance or entity only runs on a single host instance at a time. If a task hub is configured with three partitions (and therefore three control queues), orchestration instances and entities can be load-balanced across all three lease-holding host instances. Additional VMs can be added to increase capacity for activity function execution.

The following diagram illustrates how the Azure Functions host interacts with the storage entities in a scaled out environment.



As shown in the previous diagram, all VMs compete for messages on the work-item queue. However, only three VMs can acquire messages from control queues, and each VM locks a single control queue.

Orchestration instances and entities are distributed across all control queue instances. The distribution is done by hashing the instance ID of the orchestration or the entity name and key pair. Orchestration instance IDs by default are random GUIDs, ensuring that instances are equally distributed across all control queues.

Generally speaking, orchestrator functions are intended to be lightweight and should not require large amounts of computing power. It is therefore not necessary to create a large number of control queue partitions to get great throughput for orchestrations. Most of the heavy work should be done in stateless activity functions, which can be scaled out infinitely.

Extended sessions

Extended sessions is a [caching mechanism](#) that keeps orchestrations and entities in memory even after they finish processing messages. The typical effect of enabling extended sessions is reduced I/O against the underlying durable store and overall improved throughput.

You can enable extended sessions by setting `durableTask/extendedSessionsEnabled` to `true` in the `host.json` file. The `durableTask/extendedSessionIdleTimeoutInSeconds` setting can be used to control how long an idle session will be held in memory:

Functions 2.0

```
{  
  "extensions": {  
    "durableTask": {  
      "extendedSessionsEnabled": true,  
      "extendedSessionIdleTimeoutInSeconds": 30  
    }  
  }  
}
```

Functions 1.0

```
{  
  "durableTask": {  
    "extendedSessionsEnabled": true,  
    "extendedSessionIdleTimeoutInSeconds": 30  
  }  
}
```

There are two potential downsides of this setting to be aware of:

1. There's an overall increase in function app memory usage because idle instances are not unloaded from memory as quickly.
2. There can be an overall decrease in throughput if there are many concurrent, distinct, short-lived orchestrator or entity function executions.

As an example, if `durableTask/extendedSessionIdleTimeoutInSeconds` is set to 30 seconds, then a short-lived orchestrator or entity function episode that executes in less than 1 second still occupies memory for 30 seconds. It also counts against the `durableTask/maxConcurrentOrchestratorFunctions` quota mentioned previously, potentially preventing other orchestrator or entity functions from running.

The specific effects of extended sessions on orchestrator and entity functions are described in the next sections.

NOTE

Extended sessions are currently only supported in .NET languages, like C# or F#. Setting `extendedSessionsEnabled` to `true` for other platforms can lead to runtime issues, such as silently failing to execute activity and orchestration-triggered functions.

Orchestrator function replay

As mentioned previously, orchestrator functions are replayed using the contents of the **History** table. By default, the orchestrator function code is replayed every time a batch of messages are dequeued from a control queue. Even if you are using the fan-out, fan-in pattern and are awaiting for all tasks to complete (for example, using `Task.WhenAll()` in .NET, `context.df.Task.all()` in JavaScript, or `context.task_all()` in Python), there will be replays that occur as batches of task responses are processed over time. When extended sessions are enabled, orchestrator function instances are held in memory longer and new messages can be processed without a full history replay.

The performance improvement of extended sessions is most often observed in the following situations:

- When there are a limited number of orchestration instances running concurrently.
- When orchestrations have large number of sequential actions (for example, hundreds of activity function calls) that complete quickly.
- When orchestrations fan-out and fan-in a large number of actions that complete around the same time.
- When orchestrator functions need to process large messages or do any CPU-intensive data processing.

In all other situations, there is typically no observable performance improvement for orchestrator functions.

NOTE

These settings should only be used after an orchestrator function has been fully developed and tested. The default aggressive replay behavior can be useful for detecting [orchestrator function code constraints](#) violations at development time, and is therefore disabled by default.

Performance targets

The following table shows the expected *maximum* throughput numbers for the scenarios described in the [Performance Targets](#) section of the [Performance and Scale](#) article.

"Instance" refers to a single instance of an orchestrator function running on a single small ([A1](#)) VM in Azure App Service. In all cases, it is assumed that [extended sessions](#) are enabled. Actual results may vary depending on the CPU or I/O work performed by the function code.

SCENARIO	MAXIMUM THROUGHPUT
Sequential activity execution	5 activities per second, per instance
Parallel activity execution (fan-out)	100 activities per second, per instance
Parallel response processing (fan-in)	150 responses per second, per instance
External event processing	50 events per second, per instance
Entity operation processing	64 operations per second

If you are not seeing the throughput numbers you expect and your CPU and memory usage appears healthy, check to see whether the cause is related to [the health of your storage account](#). The Durable Functions extension can put significant load on an Azure Storage account and sufficiently high loads may result in storage account throttling.

TIP

In some cases you can significantly increase the throughput of external events, activity fan-in, and entity operations by increasing the value of the `controlQueueBufferThreshold` setting in `host.json`. Increasing this value beyond its default causes the Durable Task Framework storage provider to use more memory to prefetch these events more aggressively, reducing delays associated with dequeuing messages from the Azure Storage control queues. For more information, see the [host.json](#) reference documentation.

High throughput processing

The architecture of the Azure Storage backend puts certain limitations on the maximum theoretical performance and scalability of Durable Functions. If your testing shows that Durable Functions on Azure Storage won't meet your throughput requirements, you should consider instead using the [Netherite storage provider for Durable](#)

Functions.

To compare the achievable throughput for various basic scenarios, see the section [Basic Scenarios](#) of the Netherite storage provider documentation.

The Netherite storage backend was designed and developed by [Microsoft Research](#). It uses [Azure Event Hubs](#) and the [FASTER](#) database technology on top of [Azure Page Blobs](#). The design of Netherite enables significantly higher-throughput processing of orchestrations and entities compared to other providers. In some benchmark scenarios, throughput was shown to increase by more than an order of magnitude when compared to the default Azure Storage provider.

For more information on the supported storage providers for Durable Functions and how they compare, see the [Durable Functions storage providers](#) documentation.

Next steps

[Learn about disaster recovery and geo-distribution](#)

Create Durable Functions using the Azure portal

10/5/2022 • 5 minutes to read • [Edit Online](#)

The [Durable Functions](#) extension for Azure Functions is provided in the NuGet package `Microsoft.Azure.WebJobs.Extensions.DurableTask`. This extension must be installed in your function app. This article shows how to install this package so that you can develop durable functions in the Azure portal.

NOTE

- If you are developing durable functions in C#, you should instead consider [Visual Studio 2019 development](#).
- If you are developing durable functions in JavaScript, you should instead consider [Visual Studio Code development](#).

Create a function app

You must have a function app to host the execution of any function. A function app lets you group your functions as a logical unit for easier management, deployment, scaling, and sharing of resources. You can create a .NET or JavaScript app.

1. From the Azure portal menu or the [Home](#) page, select **Create a resource**.
2. In the **New** page, select **Compute > Function App**.
3. On the **Basics** page, use the function app settings as specified in the following table:

SETTING	SUGGESTED VALUE	DESCRIPTION
Subscription	Your subscription	The subscription under which you'll create your new function app.
Resource Group	<i>myResourceGroup</i>	Name for the new resource group in which you'll create your function app.
Function App name	Globally unique name	Name that identifies your new function app. Valid characters are <code>a-z</code> (case insensitive), <code>0-9</code> , and <code>-</code> .
Publish	Code	Option to publish code files or a Docker container.
Runtime stack	Preferred language	Choose a runtime that supports your favorite function programming language. In-portal editing is only available for JavaScript, PowerShell, TypeScript, and C# script. C# class library, Java, and Python functions must be developed locally .
Version	Version number	Choose the version of your installed runtime.

SETTING	SUGGESTED VALUE	DESCRIPTION
Region	Preferred region	Select a region that's near you or near other services that your functions can access.

4. Select **Next : Hosting**. On the **Hosting** page, enter the following settings:

SETTING	SUGGESTED VALUE	DESCRIPTION
Storage account	Globally unique name	Create a storage account used by your function app. Storage account names must be between 3 and 24 characters in length and can contain numbers and lowercase letters only. You can also use an existing account, which must meet the storage account requirements .
Operating system	Windows	An operating system is pre-selected for you based on your runtime stack selection, but you can change the setting if necessary. In-portal editing is only supported on Windows.
Plan	Consumption (Serverless)	Hosting plan that defines how resources are allocated to your function app. In the default Consumption plan, resources are added dynamically as required by your functions. In this serverless hosting, you pay only for the time your functions run. When you run in an App Service plan, you must manage the scaling of your function app .

5. Select **Next : Monitoring**. On the **Monitoring** page, enter the following settings:

SETTING	SUGGESTED VALUE	DESCRIPTION
Application Insights	Default	Creates an Application Insights resource of the same <i>App name</i> in the nearest supported region. By expanding this setting or selecting Create new , you can change the Application Insights name or select a different region in an Azure geography where you want to store your data.

6. Select **Review + create** to review the app configuration selections.

7. On the **Review + create** page, review your settings, and then select **Create** to provision and deploy the function app.

8. Select the **Notifications** icon in the upper-right corner of the portal and watch for the **Deployment succeeded** message.

9. Select **Go to resource** to view your new function app. You can also select **Pin to dashboard**. Pinning makes it easier to return to this function app resource from your dashboard.

The screenshot shows the Azure Notifications page. At the top, there are several icons: a blue square, a mail icon, a clipboard icon, a bell icon, a gear icon, a question mark icon, and a user profile icon. To the right of the icons is the text "MICROSOFT (MICROSOFT.ONMI...)" and a close button (X). Below the header, the title "Notifications" is displayed. On the left, there is a link "More events in the activity log →". On the right, there is a "Dismiss all" button with a dropdown arrow. A deployment message is shown: "Deployment succeeded" with a checkmark icon. The message text is: "Deployment 'Microsoft.Web-FunctionApp-Portal-57943c9a-a5b9' to resource group 'myResourceGroup' was successful." Below the message are two buttons: "Go to resource" (highlighted with a red border) and "Pin to dashboard". To the right of the buttons is the timestamp "a few seconds ago".

By default, the function app created uses version 2.x of the Azure Functions runtime. The Durable Functions extension works on both versions 1.x and 2.x of the Azure Functions runtime in C#, and version 2.x in JavaScript. However, templates are only available when targeting version 2.x of the runtime regardless of the chosen language.

Install the durable-functions npm package (JavaScript only)

If you are creating JavaScript Durable Functions, you'll need to install the [durable-functions npm package](#):

1. From your function app's page, select **Advanced Tools** under **Development Tools** in the left pane.

The screenshot shows the "myfunctionapp | Advanced Tools" page. The left sidebar has a search bar and a list of tools: "Development Tools" (Console, Advanced Tools, App Service Editor (Preview), Resource explorer, Extensions), "API" (API Management, API definition, CORS). The "Advanced Tools" item is highlighted with a red box. The main content area has a title "Advanced Tools" with a pencil icon. Below the title is a description: "Advanced Tools provides a collection of developer oriented tools and extensibility points for your App Service Apps. [Learn more](#)". There is also a "Go →" button.

2. In the **Advanced Tools** page, select **Go**.
3. Inside the Kudu console, select **Debug console**, and then **CMD**.

The screenshot shows the Kudu interface with the 'Debug console' dropdown menu open. The 'CMD' option is selected and highlighted with a red box. Other options like 'PowerShell' are visible but not selected.

Environment

Build	79.11121.3655.0 (935294dece)
Azure App Service	78.0.8598.95 (rd_websites_stable.181101-1149)
Site up time	00:00:45:44
Site folder	D:\home
Temp folder	D:\local\Temp\

4. Your function app's file directory structure should display. Navigate to the `site/wwwroot` folder. From there, you can upload a `package.json` file by dragging and dropping it into the file directory window. A sample `package.json` is below:

```
{  
  "dependencies": {  
    "durable-functions": "^1.3.1"  
  }  
}
```

The screenshot shows the Kudu interface with the 'Debug console' dropdown menu open. The 'CMD' option is selected and highlighted with a red box. Other options like 'PowerShell' are visible but not selected.

Environment

Build	79.11121.3655.0 (935294dece)
Azure App Service	78.0.8598.95 (rd_websites_stable.181101-1149)
Site up time	00:00:45:44
Site folder	D:\home
Temp folder	D:\local\Temp\

5. Once your `package.json` is uploaded, run the `npm install` command from the Kudu Remote Execution Console.

The screenshot shows the Kudu Remote Execution Console. The command `npm install` is highlighted with a red box at the bottom of the terminal window. The console output shows the command being run in the `D:\home\site\wwwroot` directory.

```
Kudu Remote Execution Console  
Type 'exit' then hit 'enter' to get a new CMD process.  
Type 'cls' to clear the console  
  
Microsoft Windows [Version 10.0.14393]  
(c) 2016 Microsoft Corporation. All rights reserved.  
  
D:\home>  
D:\home\site>  
D:\home\site\wwwroot>npm install
```

Create an orchestrator function

1. In your function app, select **Functions** from the left pane, and then select **Add** from the top menu.
2. In the search field of the **New Function** page, enter `durable`, and then choose the **Durable Functions HTTP starter** template.

The screenshot shows the Azure Functions blade for a function app named 'myfunctionapp'. On the left, there's a navigation sidebar with 'Functions' selected. In the main area, a search bar at the top right contains the text 'durable'. Below it, under the 'Templates' tab, the 'Durable Functions HTTP starter' template is highlighted with a red box. This template is described as a function that triggers whenever it receives an HTTP request to execute an orchestrator function. To its right is the 'Durable Functions activity' template, which is described as a function that runs whenever an Activity is called by an orchestrator function.

- For the **New Function** name, enter `HttpStart`, and then select **Create Function**.

The function created is used to start the orchestration.

- Create another function in the function app, this time by using the **Durable Functions orchestrator** template. Name your new orchestration function `HelloSequence`.

- Create a third function named `Hello` by using the **Durable Functions activity** template.

Test the durable function orchestration

- Go back to the `HttpStart` function, choose **Get function Url**, and select the **Copy to clipboard** icon to copy the URL. You use this URL to start the `HelloSequence` function.
- Use an HTTP tool like Postman or cURL to send a POST request to the URL that you copied. The following example is a cURL command that sends a POST request to the durable function:

```
curl -X POST https://{{your-function-app-name}}.azurewebsites.net/api/orchestrators/{{functionName}} --header "Content-Length: 0"
```

In this example, `{{your-function-app-name}}` is the domain that is the name of your function app, and `{{functionName}}` is the `HelloSequence` orchestrator function. The response message contains a set of URI endpoints that you can use to monitor and manage the execution, which looks like the following example:

```
{
  "id": "10585834a930427195479de25e0b952d",
  "statusQueryGetUri": "https://...",
  "sendEventPostUri": "https://...",
  "terminatePostUri": "https://...",
  "rewindPostUri": "https://..."
}
```

- Call the `statusQueryGetUri` endpoint URI and you see the current status of the durable function, which might look like this example:

```
{  
    "runtimeStatus": "Running",  
    "input": null,  
    "output": null,  
    "createdTime": "2017-12-01T05:37:33Z",  
    "lastUpdatedTime": "2017-12-01T05:37:36Z"  
}
```

4. Continue calling the `statusQueryGetUri` endpoint until the status changes to **Completed**, and you see a response like the following example:

```
{  
    "runtimeStatus": "Completed",  
    "input": null,  
    "output": [  
        "Hello Tokyo!",  
        "Hello Seattle!",  
        "Hello London!"  
    ],  
    "createdTime": "2017-12-01T05:38:22Z",  
    "lastUpdatedTime": "2017-12-01T05:38:28Z"  
}
```

Your first durable function is now up and running in Azure.

Next steps

[Learn about common durable function patterns](#)

Manage connections in Azure Functions

10/5/2022 • 5 minutes to read • [Edit Online](#)

Functions in a function app share resources. Among those shared resources are connections: HTTP connections, database connections, and connections to services such as Azure Storage. When many functions are running concurrently in a Consumption plan, it's possible to run out of available connections. This article explains how to code your functions to avoid using more connections than they need.

NOTE

Connection limits described in this article apply only when running in a [Consumption plan](#). However, the techniques described here may be beneficial when running on any plan.

Connection limit

The number of available connections in a Consumption plan is limited partly because a function app in this plan runs in a [sandbox environment](#). One of the restrictions that the sandbox imposes on your code is a limit on the number of outbound connections, which is currently 600 active (1,200 total) connections per instance. When you reach this limit, the functions runtime writes the following message to the logs:

`Host thresholds exceeded: Connections`. For more information, see the [Functions service limits](#).

This limit is per instance. When the [scale controller adds function app instances](#) to handle more requests, each instance has an independent connection limit. That means there's no global connection limit, and you can have much more than 600 active connections across all active instances.

When troubleshooting, make sure that you have enabled Application Insights for your function app. Application Insights lets you view metrics for your function apps like executions. For more information, see [View telemetry in Application Insights](#).

Static clients

To avoid holding more connections than necessary, reuse client instances rather than creating new ones with each function invocation. We recommend reusing client connections for any language that you might write your function in. For example, .NET clients like the [HttpClient](#), [DocumentClient](#), and Azure Storage clients can manage connections if you use a single, static client.

Here are some guidelines to follow when you're using a service-specific client in an Azure Functions application:

- *Do not* create a new client with every function invocation.
- *Do* create a single, static client that every function invocation can use.
- *Consider* creating a single, static client in a shared helper class if different functions use the same service.

Client code examples

This section demonstrates best practices for creating and using clients from your function code.

HTTP requests

- [C#](#)
- [JavaScript](#)

Here's an example of C# function code that creates a static [HttpClient](#) instance:

```
// Create a single, static HttpClient
private static HttpClient httpClient = new HttpClient();

public static async Task Run(string input)
{
    var response = await httpClient.GetAsync("https://example.com");
    // Rest of function
}
```

A common question about [HttpClient](#) in .NET is "Should I dispose of my client?" In general, you dispose of objects that implement [IDisposable](#) when you're done using them. But you don't dispose of a static client because you aren't done using it when the function ends. You want the static client to live for the duration of your application.

Azure Cosmos DB clients

- [C#](#)
- [JavaScript](#)

[CosmosClient](#) connects to an Azure Cosmos DB instance. The Azure Cosmos DB documentation recommends that you [use a singleton Azure Cosmos DB client for the lifetime of your application](#). The following example shows one pattern for doing that in a function:

```
#r "Microsoft.Azure.Cosmos"
using Microsoft.Azure.Cosmos;

private static Lazy<CosmosClient> lazyClient = new Lazy<CosmosClient>(InitializeCosmosClient);
private static CosmosClient cosmosClient => lazyClient.Value;

private static CosmosClient InitializeCosmosClient()
{
    // Perform any initialization here
    var uri = "https://youraccount.documents.azure.com:443";
    var authKey = "authKey";

    return new CosmosClient(uri, authKey);
}

public static async Task Run(string input)
{
    Container container = cosmosClient.GetContainer("database", "collection");
    MyItem item = new MyItem{ id = "myId", partitionKey = "myPartitionKey", data = "example" };
    await container.UpsertItemAsync(item);

    // Rest of function
}
```

Also, create a file named "function.proj" for your trigger and add the below content:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.Azure.Cosmos" Version="3.23.0" />
  </ItemGroup>
</Project>
```

SqlClient connections

Your function code can use the .NET Framework Data Provider for SQL Server ([SqlClient](#)) to make connections to a SQL relational database. This is also the underlying provider for data frameworks that rely on ADO.NET, such as [Entity Framework](#). Unlike [HttpClient](#) and [DocumentClient](#) connections, ADO.NET implements connection pooling by default. But because you can still run out of connections, you should optimize connections to the database. For more information, see [SQL Server Connection Pooling \(ADO.NET\)](#).

TIP

Some data frameworks, such as Entity Framework, typically get connection strings from the [ConnectionStrings](#) section of a configuration file. In this case, you must explicitly add SQL database connection strings to the [Connection strings](#) collection of your function app settings and in the [local.settings.json](#) file in your local project. If you're creating an instance of [SqlConnection](#) in your function code, you should store the connection string value in [Application settings](#) with your other connections.

Next steps

For more information about why we recommend static clients, see [Improper instantiation antipattern](#).

For more Azure Functions performance tips, see [Optimize the performance and reliability of Azure Functions](#).

Durable Functions unit testing (C#)

10/5/2022 • 6 minutes to read • [Edit Online](#)

Unit testing is an important part of modern software development practices. Unit tests verify business logic behavior and protect from introducing unnoticed breaking changes in the future. Durable Functions can easily grow in complexity so introducing unit tests will help to avoid breaking changes. The following sections explain how to unit test the three function types - Orchestration client, orchestrator, and activity functions.

NOTE

This article provides guidance for unit testing for Durable Functions apps written in C# for the .NET in-process worker and targeting Durable Functions 2.x. For more information about the differences between versions, see the [Durable Functions versions](#) article.

Prerequisites

The examples in this article require knowledge of the following concepts and frameworks:

- Unit testing
- Durable Functions
- [xUnit](#) - Testing framework
- [moq](#) - Mocking framework

Base classes for mocking

Mocking is supported via the following interface:

- [IDurableOrchestrationClient](#), [IDurableEntityClient](#) and [IDurableClient](#)
- [IDurableOrchestrationContext](#)
- [IDurableActivityContext](#)
- [IDurableEntityContext](#)

These interfaces can be used with the various trigger and bindings supported by Durable Functions. When executing your Azure Functions, the functions runtime will run your function code with a concrete implementation of these interfaces. For unit testing, you can pass in a mocked version of these interfaces to test your business logic.

Unit testing trigger functions

In this section, the unit test will validate the logic of the following HTTP trigger function for starting new orchestrations.

```

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

using System.Net.Http;
using System.Threading.Tasks;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.DurableTask;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Extensions.Logging;

namespace VSSample
{
    public static class HttpStart
    {
        [FunctionName("HttpStart")]
        public static async Task<HttpResponseMessage> Run(
            [HttpTrigger(AuthorizationLevel.Function, methods: "post", Route =
"orchestrators/{functionName}")] HttpRequestMessage req,
            [DurableClient] IDurableClient starter,
            string functionName,
            ILogger log)
        {
            // Function input comes from the request content.
            object eventData = await req.Content.ReadAsAsync<object>();
            string instanceId = await starter.StartNewAsync(functionName, eventData);

            log.LogInformation($"Started orchestration with ID = '{instanceId}'.");

            return starter.CreateCheckStatusResponse(req, instanceId);
        }
    }
}

```

The unit test task will be to verify the value of the `Retry-After` header provided in the response payload. So the unit test will mock some of `IDurableClient` methods to ensure predictable behavior.

First, we use a mocking framework ([moq](#) in this case) to mock `IDurableClient`:

```

// Mock IDurableClient
var durableClientMock = new Mock<IDurableClient>();

```

NOTE

While you can mock interfaces by directly implementing the interface as a class, mocking frameworks simplify the process in various ways. For instance, if a new method is added to the interface across minor releases, moq will not require any code changes unlike concrete implementations.

Then `StartNewAsync` method is mocked to return a well-known instance ID.

```

// Mock StartNewAsync method
durableClientMock.
    Setup(x => x.StartNewAsync(functionName, It.IsAny<object>())).
    ReturnsAsync(instanceId);

```

Next `CreateCheckStatusResponse` is mocked to always return an empty HTTP 200 response.

```
// Mock CreateCheckStatusResponse method
durableClientMock
    // Notice that even though the HttpStart function does not call
    IDurableClient.CreateCheckStatusResponse()
    // with the optional parameter returnInternalServerErrorOnFailure, moq requires the method to be set up
    // with each of the optional parameters provided. Simply use It.IsAny<> for each optional parameter
    .Setup(x => x.CreateCheckStatusResponse(It.IsAny<HttpRequestMessage>(), instanceId,
returnInternalServerErrorOnFailure: It.IsAny<bool>()))
    .Returns(new HttpResponseMessage
    {
        StatusCode = HttpStatusCode.OK,
        Content = new StringContent(string.Empty),
        Headers =
        {
            RetryAfter = new RetryConditionHeaderValue(TimeSpan.FromSeconds(10))
        }
    });
});
```

ILogger is also mocked:

```
// Mock ILogger
var loggerMock = new Mock<	ILogger>();
```

Now the Run method is called from the unit test:

```
// Call Orchestration trigger function
var result = await HttpStart.Run(
    new HttpRequestMessage()
    {
        Content = new StringContent("{}"),
        Encoding.UTF8,
        "application/json"),
        RequestUri = new Uri("http://localhost:7071/orchestrators/E1_HelloSequence"),
    },
    durableClientMock.Object,
    functionName,
    loggerMock.Object);
```

The last step is to compare the output with the expected value:

```
// Validate that output is not null
Assert.NotNull(result.Headers.RetryAfter);

// Validate output's Retry-After header value
Assert.Equal(TimeSpan.FromSeconds(10), result.Headers.RetryAfter.Delta);
```

After combining all steps, the unit test will have the following code:

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

namespace VSSample.Tests
{
    using System;
    using System.Net;
    using System.Net.Http;
    using System.Text;
    using System.Threading.Tasks;
    using System.Net.Http.Headers;
    using Microsoft.Azure.WebJobs.Extensions.DurableTask;
    using Microsoft.Extensions.Logging;
    using Moq;
    using Xunit;
```

```

using Xunit;

public class HttpStartTests
{
    [Fact]
    public async Task HttpStart_returns_retryafter_header()
    {
        // Define constants
        const string functionName = "SampleFunction";
        const string instanceId = "7E467BDB-213F-407A-B86A-1954053D3C24";

        // Mock TraceWriter
        var loggerMock = new Mock<ILogger>();

        // Mock DurableOrchestrationClientBase
        var clientMock = new Mock<IDurableClient>();

        // Mock StartNewAsync method
        clientMock
            .Setup(x => x.StartNewAsync(functionName, It.IsAny<string>(), It.IsAny<object>()))
            .ReturnsAsync(instanceId);

        // Mock CreateCheckStatusResponse method
        clientMock
            .Setup(x => x.CreateCheckStatusResponse(It.IsAny<HttpRequestMessage>(), instanceId, false))
            .>Returns(new HttpResponseMessage
            {
                StatusCode = HttpStatusCode.OK,
                Content = new StringContent(string.Empty),
                Headers =
                {
                    RetryAfter = new RetryConditionHeaderValue(TimeSpan.FromSeconds(10))
                }
            });
    }

    // Call Orchestration trigger function
    var result = await HttpStart.Run(
        new HttpRequestMessage()
    {
        Content = new StringContent("{}", Encoding.UTF8, "application/json"),
        RequestUri = new Uri("http://localhost:7071/orchestrators/E1_HelloSequence"),
    },
    clientMock.Object,
    functionName,
    loggerMock.Object);

    // Validate that output is not null
    Assert.NotNull(result.Headers.RetryAfter);

    // Validate output's Retry-After header value
    Assert.Equal(TimeSpan.FromSeconds(10), result.Headers.RetryAfter.Delta);
}
}
}

```

Unit testing orchestrator functions

Orchestrator functions are even more interesting for unit testing since they usually have a lot more business logic.

In this section the unit tests will validate the output of the `E1_HelloSequence` Orchestrator function:

```

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

using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.DurableTask;

namespace VSSample
{
    public static class HelloSequence
    {
        [FunctionName("E1_HelloSequence")]
        public static async Task<List<string>> Run(
            [OrchestrationTrigger] IDurableOrchestrationContext context)
        {
            var outputs = new List<string>();

            outputs.Add(await context.CallActivityAsync<string>("E1_SayHello", "Tokyo"));
            outputs.Add(await context.CallActivityAsync<string>("E1_SayHello", "Seattle"));
            outputs.Add(await context.CallActivityAsync<string>("E1_SayHello_DirectInput", "London"));

            // returns ["Hello Tokyo!", "Hello Seattle!", "Hello London!"]
            return outputs;
        }

        [FunctionName("E1_SayHello")]
        public static string SayHello([ActivityTrigger] IDurableActivityContext context)
        {
            string name = context.GetInput<string>();
            return $"Hello {name}!";
        }

        [FunctionName("E1_SayHello_DirectInput")]
        public static string SayHelloDirectInput([ActivityTrigger] string name)
        {
            return $"Hello {name}!";
        }
    }
}

```

The unit test code will start with creating a mock:

```
var durableOrchestrationContextMock = new Mock<IDurableOrchestrationContext>();
```

Then the activity method calls will be mocked:

```

durableOrchestrationContextMock.Setup(x => x.CallActivityAsync<string>("E1_SayHello",
    "Tokyo")).ReturnsAsync("Hello Tokyo!");
durableOrchestrationContextMock.Setup(x => x.CallActivityAsync<string>("E1_SayHello",
    "Seattle")).ReturnsAsync("Hello Seattle!");
durableOrchestrationContextMock.Setup(x => x.CallActivityAsync<string>("E1_SayHello",
    "London")).ReturnsAsync("Hello London!");

```

Next the unit test will call `HelloSequence.Run` method:

```
var result = await HelloSequence.Run(durableOrchestrationContextMock.Object);
```

And finally the output will be validated:

```
Assert.Equal(3, result.Count);
Assert.Equal("Hello Tokyo!", result[0]);
Assert.Equal("Hello Seattle!", result[1]);
Assert.Equal("Hello London!", result[2]);
```

After combining all steps, the unit test will have the following code:

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

namespace VSSample.Tests
{
    using System.Threading.Tasks;
    using Microsoft.Azure.WebJobs.Extensions.DurableTask;
    using Moq;
    using Xunit;

    public class HelloSequenceTests
    {
        [Fact]
        public async Task Run_returns_multiple_greetings()
        {
            var mockContext = new Mock<IDurableOrchestrationContext>();
            mockContext.Setup(x => x.CallActivityAsync<string>("E1_SayHello", "Tokyo")).ReturnsAsync("Hello
Tokyo!");
            mockContext.Setup(x => x.CallActivityAsync<string>("E1_SayHello",
"Seattle")).ReturnsAsync("Hello Seattle!");
            mockContext.Setup(x => x.CallActivityAsync<string>("E1_SayHello_DirectInput",
"London")).ReturnsAsync("Hello London!");

            var result = await HelloSequence.Run(mockContext.Object);

            Assert.Equal(3, result.Count);
            Assert.Equal("Hello Tokyo!", result[0]);
            Assert.Equal("Hello Seattle!", result[1]);
            Assert.Equal("Hello London!", result[2]);
        }
    }
}
```

Unit testing activity functions

Activity functions can be unit tested in the same way as non-durable functions.

In this section the unit test will validate the behavior of the `E1_SayHello` Activity function:

```

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

using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.DurableTask;

namespace VSSample
{
    public static class HelloSequence
    {
        [FunctionName("E1_HelloSequence")]
        public static async Task<List<string>> Run(
            [OrchestrationTrigger] IDurableOrchestrationContext context)
        {
            var outputs = new List<string>();

            outputs.Add(await context.CallActivityAsync<string>("E1_SayHello", "Tokyo"));
            outputs.Add(await context.CallActivityAsync<string>("E1_SayHello", "Seattle"));
            outputs.Add(await context.CallActivityAsync<string>("E1_SayHello_DirectInput", "London"));

            // returns ["Hello Tokyo!", "Hello Seattle!", "Hello London!"]
            return outputs;
        }

        [FunctionName("E1_SayHello")]
        public static string SayHello([ActivityTrigger] IDurableActivityContext context)
        {
            string name = context.GetInput<string>();
            return $"Hello {name}!";
        }

        [FunctionName("E1_SayHello_DirectInput")]
        public static string SayHelloDirectInput([ActivityTrigger] string name)
        {
            return $"Hello {name}!";
        }
    }
}

```

And the unit tests will verify the format of the output. The unit tests can use the parameter types directly or mock `IDurableActivityContext` class:

```
// Copyright (c) .NET Foundation. All rights reserved.  
// Licensed under the MIT License. See LICENSE in the project root for license information.  
  
namespace VSSample.Tests  
{  
    using Microsoft.Azure.WebJobs.Extensions.DurableTask;  
    using Xunit;  
    using Moq;  
  
    public class HelloSequenceActivityTests  
    {  
        [Fact]  
        public void SayHello_returns_greeting()  
        {  
            var durableActivityContextMock = new Mock<IDurableActivityContext>();  
            durableActivityContextMock.Setup(x => x.GetInput<string>()).Returns("John");  
            var result = HelloSequence.SayHello(durableActivityContextMock.Object);  
            Assert.Equal("Hello John!", result);  
        }  
  
        [Fact]  
        public void SayHello_returns_greeting_direct_input()  
        {  
            var result = HelloSequence.SayHelloDirectInput("John");  
            Assert.Equal("Hello John!", result);  
        }  
    }  
}
```

Next steps

[Learn more about xUnit](#)

[Learn more about moq](#)

How to run Durable Functions as WebJobs

10/5/2022 • 6 minutes to read • [Edit Online](#)

By default, Durable Functions uses the Azure Functions runtime to host orchestrations. However, there may be certain scenarios where you need more control over the code that listens for events. This article shows you how to implement your orchestration using the WebJobs SDK. To see a more detailed comparison between Functions and WebJobs, see [Compare Functions and WebJobs](#).

Azure Functions and the Durable Functions extension are built on the [WebJobs SDK](#). The job host in the WebJobs SDK is the runtime in Azure Functions. If you need to control behavior in ways not possible in Azure Functions, you can develop and run Durable Functions by using the WebJobs SDK yourself.

In version 3.x of the WebJobs SDK, the host is an implementation of `IHost`, and in version 2.x you use the `JobHost` object.

The chaining Durable Functions sample is available in a WebJobs SDK 2.x version: download or clone the [Durable Functions repository](#), and checkout *v1* branch and go to the *samples\webjobssdk\chaining* folder.

Prerequisites

This article assumes you're familiar with the basics of the WebJobs SDK, C# class library development for Azure Functions, and Durable Functions. If you need an introduction to these topics, see the following resources:

- [Get started with the WebJobs SDK](#)
- [Create your first function using Visual Studio](#)
- [Durable Functions](#)

To complete the steps in this article:

- [Install Visual Studio 2019](#) with the **Azure development** workload.

If you already have Visual Studio, but don't have that workload, add the workload by selecting **Tools > Get Tools and Features**.

(You can use [Visual Studio Code](#) instead, but some of the instructions are specific to Visual Studio.)

- Install and run an [Azure Storage Emulator](#). An alternative is to update the *App.config* file with a real Azure Storage connection string.

WebJobs SDK versions

This article explains how to develop a WebJobs SDK 2.x project (equivalent to Azure Functions version 1.x). For information about version 3.x, see [WebJobs SDK 3.x](#) later in this article.

Create a console app

To run Durable Functions as WebJobs, you must first create a console app. A WebJobs SDK project is just a console app project with the appropriate NuGet packages installed.

In the Visual Studio **New Project** dialog box, select **Windows Classic Desktop > Console App (.NET Framework)**. In the project file, the `TargetFrameworkVersion` should be `v4.6.1`.

Visual Studio also has a WebJob project template, which you can use by selecting **Cloud > Azure WebJob**.

(.NET Framework). This template installs many packages, some of which you might not need.

Install NuGet packages

You need NuGet packages for the WebJobs SDK, core bindings, the logging framework, and the Durable Task extension. Here are **Package Manager Console** commands for those packages, with the latest stable version numbers as of the date this article was written:

```
Install-Package Microsoft.Azure.WebJobs.Extensions -version 2.2.0
Install-Package Microsoft.Extensions.Logging -version 2.0.1
Install-Package Microsoft.Azure.WebJobs.Extensions.DurableTask -version 1.8.7
```

You also need logging providers. The following commands install the Azure Application Insights provider and the `ConfigurationManager`. The `ConfigurationManager` lets you get the Application Insights instrumentation key from app settings.

```
Install-Package Microsoft.Azure.WebJobs.Logging.ApplicationInsights -version 2.2.0
Install-Package System.Configuration.ConfigurationManager -version 4.4.1
```

The following command installs the console provider:

```
Install-Package Microsoft.Extensions.Logging.Console -version 2.0.1
```

JobHost code

Having created the console app and installed the NuGet packages you need, you're ready to use Durable Functions. You do so by using JobHost code.

To use the Durable Functions extension, call `UseDurableTask` on the `JobHostConfiguration` object in your `Main` method:

```
var config = new JobHostConfiguration();
config.UseDurableTask(new DurableTaskExtension
{
    HubName = "MyTaskHub",
});
```

For a list of properties that you can set in the `DurableTaskExtension` object, see [host.json](#).

The `Main` method is also the place to set up logging providers. The following example configures the console and Application Insights providers.

```

static void Main(string[] args)
{
    using (var loggerFactory = new LoggerFactory())
    {
        var config = new JobHostConfiguration();

        config.DashboardConnectionString = "";

        var instrumentationKey =
            ConfigurationManager.AppSettings["APPINSIGHTS_INSTRUMENTATIONKEY"];

        config.LoggerFactory = loggerFactory
            .AddApplicationInsights(instrumentationKey, null)
            .AddConsole();

        config.UseTimers();
        config.UseDurableTask(new DurableTaskExtension
        {
            HubName = "MyTaskHub",
        });
        var host = new JobHost(config);
        host.RunAndBlock();
    }
}

```

Functions

Durable Functions in the context of WebJobs differs somewhat from Durable Functions in the context of Azure Functions. It's important to be aware of the differences as you write your code.

The WebJobs SDK doesn't support the following Azure Functions features:

- [FunctionName attribute](#)
- [HTTP trigger](#)
- [Durable Functions HTTP management API](#)

FunctionName attribute

In a WebJobs SDK project, the method name of a function is the function name. The `FunctionName` attribute is used only in Azure Functions.

HTTP trigger

The WebJobs SDK does not have an HTTP trigger. The sample project's orchestration client uses a timer trigger:

```

public static async Task CronJob(
    [TimerTrigger("0 */2 * * * *")] TimerInfo timer,
    [OrchestrationClient] DurableOrchestrationClient client,
    ILogger logger)
{
    ...
}

```

HTTP management API

Because it has no HTTP trigger, the WebJobs SDK has no [HTTP management API](#).

In a WebJobs SDK project, you can call methods on the orchestration client object, instead of by sending HTTP requests. The following methods correspond to the three tasks you can do with the HTTP management API:

- `GetStatusAsync`
- `RaiseEventAsync`

- `TerminateAsync`

The orchestration client function in the sample project starts the orchestrator function, and then goes into a loop that calls `GetStatusAsync` every 2 seconds:

```
string instanceId = await client.StartNewAsync(nameof(HelloSequence), input: null);
logger.LogInformation($"Started new instance with ID = {instanceId}.");

DurableOrchestrationStatus status;
while (true)
{
    status = await client.GetStatusAsync(instanceId);
    logger.LogInformation($"Status: {status.RuntimeStatus}, Last update: {status.LastUpdatedTime}.");

    if (status.RuntimeStatus == OrchestrationRuntimeStatus.Completed ||
        status.RuntimeStatus == OrchestrationRuntimeStatus.Failed ||
        status.RuntimeStatus == OrchestrationRuntimeStatus.Terminated)
    {
        break;
    }

    await Task.Delay(TimeSpan.FromSeconds(2));
}
```

Run the sample

You've got Durable Functions set up to run as a WebJob, and you now have an understanding of how this will differ from running Durable Functions as standalone Azure Functions. At this point, seeing it work in a sample might be helpful.

This section provides an overview of how to run the [sample project](#). For detailed instructions that explain how to run a WebJobs SDK project locally and deploy it to an Azure WebJob, see [Get started with the WebJobs SDK](#).

Run locally

1. Make sure the Storage emulator is running (see [Prerequisites](#)).
2. If you want to see logs in Application Insights when you run the project locally:
 - a. Create an Application Insights resource, and use the **General** app type for it.
 - b. Save the instrumentation key in the *App.config* file.
3. Run the project.

Run in Azure

1. Create a web app and a storage account.
2. In the web app, save the storage connection string in an app setting named `AzureWebJobsStorage`.
3. Create an Application Insights resource, and use the **General** app type for it.
4. Save the instrumentation key in an app setting named `APPINSIGHTS_INSTRUMENTATIONKEY`.
5. Deploy as a WebJob.

WebJobs SDK 3.x

This article explains how to develop a WebJobs SDK 2.x project. If you're developing a [WebJobs SDK 3.x](#) project, this section helps you understand the differences.

The main change introduced is the use of .NET Core instead of .NET Framework. To create a WebJobs SDK 3.x

project, the instructions are the same, with these exceptions:

1. Create a .NET Core console app. In the Visual Studio **New Project** dialog box, select **.NET Core > Console App (.NET Core)**. The project file specifies that `TargetFramework` is `netcoreapp2.x`.
2. Choose the release version WebJobs SDK 3.x of the following packages:
 - `Microsoft.Azure.WebJobs.Extensions`
 - `Microsoft.Azure.WebJobs.Extensions.Storage`
 - `Microsoft.Azure.WebJobs.Logging.ApplicationInsights`
3. Set the storage connection string and the Application Insights instrumentation key in an `appsettings.json` file, by using the .NET Core configuration framework. Here's an example:

```
{  
    "AzureWebJobsStorage": "<replace with storage connection string>",  
    "APPINSIGHTS_INSTRUMENTATIONKEY": "<replace with Application Insights instrumentation key>"  
}
```

4. Change the `Main` method code to do this. Here's an example:

```
static void Main(string[] args)  
{  
    var hostBuilder = new HostBuilder()  
        .ConfigureWebJobs(config =>  
    {  
        config.AddAzureStorageCoreServices();  
        config.AddAzureStorage();  
        config.AddTimers();  
        config.AddDurableTask(options =>  
        {  
            options.HubName = "MyTaskHub";  
            options.AzureStorageConnectionStringName = "AzureWebJobsStorage";  
        });  
    })  
        .ConfigureLogging((context, logging) =>  
    {  
        logging.AddConsole();  
        logging.AddApplicationInsights(config =>  
        {  
            config.InstrumentationKey = context.Configuration["APPINSIGHTS_INSTRUMENTATIONKEY"];  
        });  
    })  
        .UseConsoleLifetime();  
  
    var host = hostBuilder.Build();  
  
    using (host)  
    {  
        host.Run();  
    }  
}
```

Next steps

To learn more about the WebJobs SDK, see [How to use the WebJobs SDK](#).

Developer's guide to durable entities in .NET

10/5/2022 • 14 minutes to read • [Edit Online](#)

In this article, we describe the available interfaces for developing durable entities with .NET in detail, including examples and general advice.

Entity functions provide serverless application developers with a convenient way to organize application state as a collection of fine-grained entities. For more detail about the underlying concepts, see the [Durable Entities: Concepts](#) article.

We currently offer two APIs for defining entities:

- The **class-based syntax** represents entities and operations as classes and methods. This syntax produces easily readable code and allows operations to be invoked in a type-checked manner through interfaces.
- The **function-based syntax** is a lower-level interface that represents entities as functions. It provides precise control over how the entity operations are dispatched, and how the entity state is managed.

This article focuses primarily on the class-based syntax, as we expect it to be better suited for most applications. However, the **function-based syntax** may be appropriate for applications that wish to define or manage their own abstractions for entity state and operations. Also, it may be appropriate for implementing libraries that require genericity not currently supported by the class-based syntax.

NOTE

The class-based syntax is just a layer on top of the function-based syntax, so both variants can be used interchangeably in the same application.

Defining entity classes

The following example is an implementation of a `Counter` entity that stores a single value of type `integer`, and offers four operations `Add`, `Reset`, `Get`, and `Delete`.

```
[JsonObject(MemberSerialization.OptIn)]
public class Counter
{
    [JsonProperty("value")]
    public int Value { get; set; }

    public void Add(int amount)
    {
        this.Value += amount;
    }

    public Task Reset()
    {
        this.Value = 0;
        return Task.CompletedTask;
    }

    public Task<int> Get()
    {
        return Task.FromResult(this.Value);
    }

    public void Delete()
    {
        Entity.Current.DeleteState();
    }

    [FunctionName(nameof(Counter))]
    public static Task Run([EntityTrigger] IDurableEntityContext ctx)
        => ctx.DispatchAsync<Counter>();
}
```

The `Run` function contains the boilerplate required for using the class-based syntax. It must be a *static* Azure Function. It executes once for each operation message that is processed by the entity. When `DispatchAsync<T>` is called and the entity isn't already in memory, it constructs an object of type `T` and populates its fields from the last persisted JSON found in storage (if any). Then it invokes the method with the matching name.

NOTE

The state of a class-based entity is **created implicitly** before the entity processes an operation, and can be **deleted explicitly** in an operation by calling `Entity.Current.DeleteState()`.

Class Requirements

Entity classes are POCOs (plain old CLR objects) that require no special superclasses, interfaces, or attributes. However:

- The class must be constructible (see [Entity construction](#)).
- The class must be JSON-serializable (see [Entity serialization](#)).

Also, any method that is intended to be invoked as an operation must satisfy additional requirements:

- An operation must have at most one argument, and must not have any overloads or generic type arguments.
- An operation meant to be called from an orchestration using an interface must return `Task` or `Task<T>`.
- Arguments and return values must be serializable values or objects.

What can operations do?

All entity operations can read and update the entity state, and changes to the state are automatically persisted to storage. Moreover, operations can perform external I/O or other computations, within the general limits common to all Azure Functions.

Operations also have access to functionality provided by the `Entity.Current` context:

- `EntityName` : the name of the currently executing entity.
- `EntityKey` : the key of the currently executing entity.
- `EntityId` : the ID of the currently executing entity (includes name and key).
- `SignalEntity` : sends a one-way message to an entity.
- `CreateNewOrchestration` : starts a new orchestration.
- `DeleteState` : deletes the state of this entity.

For example, we can modify the counter entity so it starts an orchestration when the counter reaches 100 and passes the entity ID as an input argument:

```
public void Add(int amount)
{
    if (this.Value < 100 && this.Value + amount >= 100)
    {
        Entity.Current.StartNewOrchestration("MilestoneReached", Entity.Current.EntityId);
    }
    this.Value += amount;
}
```

Accessing entities directly

Class-based entities can be accessed directly, using explicit string names for the entity and its operations. We provide some examples below; for a deeper explanation of the underlying concepts (such as signals vs. calls) see the discussion in [Access entities](#).

NOTE

Where possible, we recommend [Accessing entities through interfaces](#), because it provides more type checking.

Example: client signals entity

The following Azure Http Function implements a DELETE operation using REST conventions. It sends a delete signal to the counter entity whose key is passed in the URL path.

```
[FunctionName("DeleteCounter")]
public static async Task<HttpResponseMessage> DeleteCounter(
    [HttpTrigger(AuthorizationLevel.Function, "delete", Route = "Counter/{entityKey}")] HttpRequestMessage req,
    [DurableClient] IDurableEntityClient client,
    string entityKey)
{
    var entityId = new EntityId("Counter", entityKey);
    await client.SignalEntityAsync(entityId, "Delete");
    return req.CreateResponse(HttpStatusCode.Accepted);
}
```

Example: client reads entity state

The following Azure Http Function implements a GET operation using REST conventions. It reads the current state of the counter entity whose key is passed in the URL path.

```
[FunctionName("GetCounter")]
public static async Task<HttpResponseMessage> GetCounter(
    [HttpTrigger(AuthorizationLevel.Function, "get", Route = "Counter/{entityKey}")] HttpRequestMessage req,
    [DurableClient] IDurableEntityClient client,
    string entityKey)
{
    var entityId = new EntityId("Counter", entityKey);
    var state = await client.ReadEntityStateAsync<Counter>(entityId);
    return req.CreateResponse(state);
}
```

NOTE

The object returned by `ReadEntityStateAsync` is just a local copy, that is, a snapshot of the entity state from some earlier point in time. In particular, it may be stale, and modifying this object has no effect on the actual entity.

Example: orchestration first signals, then calls entity

The following orchestration signals a counter entity to increment it, and then calls the same entity to read its latest value.

```
[FunctionName("IncrementThenGet")]
public static async Task<int> Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    var entityId = new EntityId("Counter", "myCounter");

    // One-way signal to the entity - does not await a response
    context.SignalEntity(entityId, "Add", 1);

    // Two-way call to the entity which returns a value - awaits the response
    int currentValue = await context.CallEntityAsync<int>(entityId, "Get");

    return currentValue;
}
```

Accessing entities through interfaces

Interfaces can be used for accessing entities via generated proxy objects. This approach ensures that the name and argument type of an operation matches what is implemented. We recommend using interfaces for accessing entities whenever possible.

For example, we can modify the counter example as follows:

```
public interface ICounter
{
    void Add(int amount);
    Task Reset();
    Task<int> Get();
    void Delete();
}

public class Counter : ICounter
{
    ...
}
```

Entity classes and entity interfaces are similar to the grains and grain interfaces popularized by [Orleans](#). For a more information about similarities and differences between Durable Entities and Orleans, see [Comparison with Orleans](#).

virtual actors.

Besides providing type checking, interfaces are useful for a better separation of concerns within the application. For example, since an entity may implement multiple interfaces, a single entity can serve multiple roles. Also, since an interface may be implemented by multiple entities, general communication patterns can be implemented as reusable libraries.

Example: client signals entity through interface

Client code can use `SignalEntityAsync< TEntityInterface >` to send signals to entities that implement `TEntityInterface`. For example:

```
[FunctionName("DeleteCounter")]
public static async Task<HttpResponseMessage> DeleteCounter(
    [HttpTrigger(AuthorizationLevel.Function, "delete", Route = "Counter/{entityKey}")] HttpRequestMessage req,
    [DurableClient] IDurableEntityClient client,
    string entityKey)
{
    var entityId = new EntityId("Counter", entityKey);
    await client.SignalEntityAsync<ICounter>(entityId, proxy => proxy.Delete());
    return req.CreateResponse(HttpStatusCode.Accepted);
}
```

In this example, the `proxy` parameter is a dynamically generated instance of `ICounter`, which internally translates the call to `Delete` into a signal.

NOTE

The `SignalEntityAsync` APIs can be used only for one-way operations. Even if an operation returns `Task<T>`, the value of the `T` parameter will always be null or `default`, not the actual result. For example, it doesn't make sense to signal the `Get` operation, as no value is returned. Instead, clients can use either `ReadStateAsync` to access the counter state directly, or can start an orchestrator function that calls the `Get` operation.

Example: orchestration first signals, then calls entity through proxy

To call or signal an entity from within an orchestration, `CreateEntityProxy` can be used, along with the interface type, to generate a proxy for the entity. This proxy can then be used to call or signal operations:

```
[FunctionName("IncrementThenGet")]
public static async Task<int> Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    var entityId = new EntityId("Counter", "myCounter");
    var proxy = context.CreateEntityProxy<ICounter>(entityId);

    // One-way signal to the entity - does not await a response
    proxy.Add(1);

    // Two-way call to the entity which returns a value - awaits the response
    int currentValue = await proxy.Get();

    return currentValue;
}
```

Implicitly, any operations that return `void` are signaled, and any operations that return `Task` or `Task<T>` are called. One can change this default behavior, and signal operations even if they return `Task`, by using the `SignalEntity< IInterfaceType >` method explicitly.

Shorter option for specifying the target

When calling or signaling an entity using an interface, the first argument must specify the target entity. The target can be specified either by specifying the entity ID, or, in cases where there's just one class that implements the entity, just the entity key:

```
context.SignalEntity<ICounter>(new EntityId(nameof(Counter), "myCounter"), ...);  
context.SignalEntity<ICounter>("myCounter", ...);
```

If only the entity key is specified and a unique implementation can't be found at runtime,

`InvalidOperationException` is thrown.

Restrictions on entity interfaces

As usual, all parameter and return types must be JSON-serializable. Otherwise, serialization exceptions are thrown at runtime.

We also enforce some additional rules:

- Entity interfaces must be defined in the same assembly as the entity class.
- Entity interfaces must only define methods.
- Entity interfaces must not contain generic parameters.
- Entity interface methods must not have more than one parameter.
- Entity interface methods must return `void`, `Task`, or `Task<T>`.

If any of these rules are violated, an `InvalidOperationException` is thrown at runtime when the interface is used as a type argument to `SignalEntity`, `SignalEntityAsync`, or `CreateEntityProxy`. The exception message explains which rule was broken.

NOTE

Interface methods returning `void` can only be signaled (one-way), not called (two-way). Interface methods returning `Task` or `Task<T>` can be either called or signalled. If called, they return the result of the operation, or re-throw exceptions thrown by the operation. However, when signalled, they do not return the actual result or exception from the operation, but just the default value.

Entity serialization

Since the state of an entity is durably persisted, the entity class must be serializable. The Durable Functions runtime uses the [Json.NET](#) library for this purpose, which supports a number of policies and attributes to control the serialization and deserialization process. Most commonly used C# data types (including arrays and collection types) are already serializable, and can easily be used for defining the state of durable entities.

For example, Json.NET can easily serialize and deserialize the following class:

```
[JsonObject(MemberSerialization = MemberSerialization.OptIn)]
public class User
{
    [JsonProperty("name")]
    public string Name { get; set; }

    [JsonProperty("yearOfBirth")]
    public int YearOfBirth { get; set; }

    [JsonProperty("timestamp")]
    public DateTime Timestamp { get; set; }

    [JsonProperty("contacts")]
    public Dictionary<Guid, Contact> Contacts { get; set; } = new Dictionary<Guid, Contact>();

    [JsonObject(MemberSerialization = MemberSerialization.OptOut)]
    public struct Contact
    {
        public string Name;
        public string Number;
    }

    ...
}
```

Serialization Attributes

In the example above, we chose to include several attributes to make the underlying serialization more visible:

- We annotate the class with `[JsonObject(MemberSerialization.OptIn)]` to remind us that the class must be serializable, and to persist only members that are explicitly marked as JSON properties.
- We annotate the fields to be persisted with `[JsonProperty("name")]` to remind us that a field is part of the persisted entity state, and to specify the property name to be used in the JSON representation.

However, these attributes aren't required; other conventions or attributes are permitted as long as they work with Json.NET. For example, one may use `[DataContract]` attributes, or no attributes at all:

```
[DataContract]
public class Counter
{
    [DataMember]
    public int Value { get; set; }
    ...
}

public class Counter
{
    public int Value;
    ...
}
```

By default, the name of the class is *not* stored as part of the JSON representation: that is, we use `TypeNameHandling.None` as the default setting. This default behavior can be overridden using `JsonObject` or `JsonProperty` attributes.

Making changes to class definitions

Some care is required when making changes to a class definition after an application has been run, because the stored JSON object may no longer match the new class definition. Still, it is often possible to deal correctly with changing data formats as long as one understands the deserialization process used by `JsonConvert.PopulateObject`.

For example, here are some examples of changes and their effect:

1. If a new property is added, which is not present in the stored JSON, it assumes its default value.
2. If a property is removed, which is present in the stored JSON, the previous content is lost.
3. If a property is renamed, the effect is as if removing the old one and adding a new one.
4. If the type of a property is changed so it can no longer be deserialized from the stored JSON, an exception is thrown.
5. If the type of a property is changed, but it can still be deserialized from the stored JSON, it will do so.

There are many options available for customizing the behavior of Json.NET. For example, to force an exception if the stored JSON contains a field that is not present in the class, specify the attribute

`JsonObject(MissingMemberHandling = MissingMemberHandling.Error)`. It is also possible to write custom code for deserialization that can read JSON stored in arbitrary formats.

Entity construction

Sometimes we want to exert more control over how entity objects are constructed. We now describe several options for changing the default behavior when constructing entity objects.

Custom initialization on first access

Occasionally we need to perform some special initialization before dispatching an operation to an entity that has never been accessed, or that has been deleted. To specify this behavior, one can add a conditional before the

`DispatchAsync` :

```
[FunctionName(nameof(Counter))]
public static Task Run([EntityTrigger] IDurableEntityContext ctx)
{
    if (!ctx.HasState)
    {
        ctx.SetState(...);
    }
    return ctx.DispatchAsync<Counter>();
}
```

Bindings in entity classes

Unlike regular functions, entity class methods don't have direct access to input and output bindings. Instead, binding data must be captured in the entry-point function declaration and then passed to the `DispatchAsync<T>` method. Any objects passed to `DispatchAsync<T>` will be automatically passed into the entity class constructor as an argument.

The following example shows how a `CloudBlobContainer` reference from the [blob input binding](#) can be made available to a class-based entity.

```

public class BlobBackedEntity
{
    [JsonIgnore]
    private readonly CloudBlobContainer container;

    public BlobBackedEntity(CloudBlobContainer container)
    {
        this.container = container;
    }

    // ... entity methods can use this.container in their implementations ...

    [FunctionName(nameof(BlobBackedEntity))]
    public static Task Run(
        [EntityTrigger] IDurableEntityContext context,
        [Blob("my-container", FileAccess.Read)] CloudBlobContainer container)
    {
        // passing the binding object as a parameter makes it available to the
        // entity class constructor
        return context.DispatchAsync<BlobBackedEntity>(container);
    }
}

```

For more information on bindings in Azure Functions, see the [Azure Functions Triggers and Bindings](#) documentation.

Dependency injection in entity classes

Entity classes support [Azure Functions Dependency Injection](#). The following example demonstrates how to register an `IHttpClientFactory` service into a class-based entity.

```

[assembly: FunctionsStartup(typeof(MyNamespace.Startup))]

namespace MyNamespace
{
    public class Startup : FunctionsStartup
    {
        public override void Configure(IFunctionsHostBuilder builder)
        {
            builder.Services.AddHttpClient();
        }
    }
}

```

The following snippet demonstrates how to incorporate the injected service into your entity class.

```

public class HttpEntity
{
    [JsonIgnore]
    private readonly HttpClient client;

    public HttpEntity(IHttpClientFactory factory)
    {
        this.client = factory.CreateClient();
    }

    public Task<int> GetAsync(string url)
    {
        using (var response = await this.client.GetAsync(url))
        {
            return (int)response.StatusCode;
        }
    }

    [FunctionName(nameof(HttpEntity))]
    public static Task Run([EntityTrigger] IDurableEntityContext ctx)
        => ctx.DispatchAsync<HttpEntity>();
}

```

NOTE

To avoid issues with serialization, make sure to exclude fields meant to store injected values from the serialization.

NOTE

Unlike when using constructor injection in regular .NET Azure Functions, the functions entry point method for class-based entities *must* be declared `static`. Declaring a non-static function entry point may cause conflicts between the normal Azure Functions object initializer and the Durable Entities object initializer.

Function-based syntax

So far we have focused on the class-based syntax, as we expect it to be better suited for most applications. However, the function-based syntax can be appropriate for applications that wish to define or manage their own abstractions for entity state and operations. Also, it may be appropriate when implementing libraries that require genericity not currently supported by the class-based syntax.

With the function-based syntax, the Entity Function explicitly handles the operation dispatch, and explicitly manages the state of the entity. For example, the following code shows the *Counter* entity implemented using the function-based syntax.

```
[FunctionName("Counter")]
public static void Counter([EntityTrigger] IDurableEntityContext ctx)
{
    switch (ctx.OperationName.ToLowerInvariant())
    {
        case "add":
            ctx.SetState(ctx.GetState<int>() + ctx.GetInput<int>());
            break;
        case "reset":
            ctx.SetState(0);
            break;
        case "get":
            ctx.Return(ctx.GetState<int>());
            break;
        case "delete":
            ctx.DeleteState();
            break;
    }
}
```

The entity context object

Entity-specific functionality can be accessed via a context object of type `IDurableEntityContext`. This context object is available as a parameter to the entity function, and via the async-local property `Entity.Current`.

The following members provide information about the current operation, and allow us to specify a return value.

- `EntityName` : the name of the currently executing entity.
- `EntityKey` : the key of the currently executing entity.
- `EntityId` : the ID of the currently executing entity (includes name and key).
- `OperationName` : the name of the current operation.
- `GetInput<TInput>()` : gets the input for the current operation.
- `Return(arg)` : returns a value to the orchestration that called the operation.

The following members manage the state of the entity (create, read, update, delete).

- `HasState` : whether the entity exists, that is, has some state.
- `GetState<TState>()` : gets the current state of the entity. If it does not already exist, it is created.
- `SetState(arg)` : creates or updates the state of the entity.
- `DeleteState()` : deletes the state of the entity, if it exists.

If the state returned by `GetState` is an object, it can be directly modified by the application code. There is no need to call `SetState` again at the end (but also no harm). If `GetState<TState>` is called multiple times, the same type must be used.

Finally, the following members are used to signal other entities, or start new orchestrations:

- `SignalEntity(EntityId, operation, input)` : sends a one-way message to an entity.
- `CreateNewOrchestration(orchestratorFunctionName, input)` : starts a new orchestration.

Next steps

[Learn about entity concepts](#)

Continuous deployment for Azure Functions

10/5/2022 • 2 minutes to read • [Edit Online](#)

You can use Azure Functions to deploy your code continuously by using [source control integration](#). Source control integration enables a workflow in which a code update triggers deployment to Azure. If you're new to Azure Functions, get started by reviewing the [Azure Functions overview](#).

Continuous deployment is a good option for projects where you integrate multiple and frequent contributions. When you use continuous deployment, you maintain a single source of truth for your code, which allows teams to easily collaborate. You can configure continuous deployment in Azure Functions from the following source code locations:

- [Azure Repos](#)
- [GitHub](#)
- [Bitbucket](#)

The unit of deployment for functions in Azure is the function app. All functions in a function app are deployed at the same time. After you enable continuous deployment, access to function code in the Azure portal is configured as *read-only* because the source of truth is set to be elsewhere.

Requirements for continuous deployment

For continuous deployment to succeed, your directory structure must be compatible with the basic folder structure that Azure Functions expects.

The code for all the functions in a specific function app is located in a root project folder that contains a host configuration file. The [host.json](#) file contains runtime-specific configurations and is in the root folder of the function app. A *bin* folder contains packages and other library files that the function app requires. Specific folder structures required by the function app depend on language:

- [C# compiled \(.csproj\)](#)
- [C# script \(.csx\)](#)
- [F# script](#)
- [Java](#)
- [JavaScript](#)
- [PowerShell](#)
- [Python](#)

In version 2.x and higher of the Functions runtime, all functions in the function app must share the same language stack.

NOTE

Continuous deployment is not yet supported for Linux apps running on a Consumption plan.

Set up continuous deployment

To configure continuous deployment for an existing function app, complete these steps. The steps demonstrate integration with a GitHub repository, but similar steps apply for Azure Repos or other source code repositories.

1. In your function app in the [Azure portal](#), select Deployment Center, select GitHub, and then select Authorize. If you've already authorized GitHub, select Continue and skip the next step.

The screenshot shows the Azure Function App Deployment Center interface. On the left, a sidebar lists various settings like Overview, Activity log, and Functions. The Deployment section has 'Deployment slots' and 'Deployment Center' selected. The main area is titled 'Deployment Center' and shows a workflow with four steps: SOURCE CONTROL, BUILD PROVIDER, CONFIGURE, and SUMMARY. Under 'Continuous Deployment (CI / CD)', there are four options: 'Azure Repos' (not highlighted), 'GitHub' (highlighted with a red box), 'Bitbucket', and 'Local Git'. The 'GitHub' option has a sub-section with the message 'Configure continuous integration with a GitHub repo.' and a status 'Not Authorized'. A blue button labeled 'Authorize' is located at the bottom right of this section. The entire 'GitHub' section is also highlighted with a red box.

2. In GitHub, select Authorize AzureAppService.

The screenshot shows the GitHub 'Authorize Azure App Service' screen. It features icons for Azure (green circle) and GitHub (black cat). Below the icons is the title 'Authorize Azure App Service'. The main content area shows 'Azure App Service by AzureAppService' requesting access to the user's GitHub account. It lists two scopes: 'Repository webhooks and services' (Admin access) and 'Repositories' (Public and private). At the bottom is a large green button labeled 'Authorize AzureAppService' which is highlighted with a red box. Below the button, a note says 'Authorizing will redirect to https://functions.azure.com'.

Enter your GitHub password and then select Continue.

3. Select one of the following build providers:

- **App Service build service**: Best when you don't need a build or if you need a generic build.
- **Azure Pipelines (Preview)**: Best when you need more control over the build. This provider currently is in preview.

Select **Continue**.

4. Configure information specific to the source control option you specified. For GitHub, you must enter or select values for **Organization**, **Repository**, and **Branch**. The values are based on the location of your code. Then, select **Continue**.

The screenshot shows the 'Code' configuration step of the deployment wizard. At the top, there's a progress bar with four steps: 'SOURCE CONTROL' (green checkmark), 'BUILD PROVIDER' (green checkmark), 'CONFIGURE' (blue circle with '3'), and 'SUMMARY' (gray circle with '4'). The 'Code' section has three dropdown menus: 'Organization' set to 'MyGitHub', 'Repository' set to 'functions-typescript-intermediate', and 'Branch' set to 'master'. Below these is a note in a blue box: 'If you can't find an organization or repository, you might need to enable additional permissions on GitHub.' At the bottom are two buttons: 'Back' and 'Continue', with 'Continue' being the one highlighted by a red box.

5. Review all details, and then select **Finish** to complete your deployment configuration.

When the process is finished, all code from the specified source is deployed to your app. At that point, changes in the deployment source trigger a deployment of those changes to your function app in Azure.

NOTE

After you configure continuous integration, you can no longer edit your source files in the Functions portal. If you originally published your code from your local computer, you may need to change the `WEBSITE_RUN_FROM_PACKAGE` setting in your function app to a value of `0`.

Next steps

[Best practices for Azure Functions](#)

Zip deployment for Azure Functions

10/5/2022 • 7 minutes to read • [Edit Online](#)

This article describes how to deploy your function app project files to Azure from a .zip (compressed) file. You learn how to do a push deployment, both by using Azure CLI and by using the REST APIs. [Azure Functions Core Tools](#) also uses these deployment APIs when publishing a local project to Azure.

Zip deployment is also an easy way to run your functions from the deployment package. To learn more, see [Run your functions from a package file in Azure](#).

Azure Functions has the full range of continuous deployment and integration options that are provided by Azure App Service. For more information, see [Continuous deployment for Azure Functions](#).

To speed up development, you may find it easier to deploy your function app project files directly from a .zip file. The .zip deployment API takes the contents of a .zip file and extracts the contents into the `wwwroot` folder of your function app. This .zip file deployment uses the same Kudu service that powers continuous integration-based deployments, including:

- Deletion of files that were left over from earlier deployments.
- Deployment customization, including running deployment scripts.
- Deployment logs.
- Syncing function triggers in a [Consumption plan](#) function app.

For more information, see the [.zip deployment reference](#).

Deployment .zip file requirements

The .zip file that you use for push deployment must contain all of the files needed to run your function.

IMPORTANT

When you use .zip deployment, any files from an existing deployment that aren't found in the .zip file are deleted from your function app.

The code for all the functions in a specific function app is located in a root project folder that contains a host configuration file. The `host.json` file contains runtime-specific configurations and is in the root folder of the function app. A `bin` folder contains packages and other library files that the function app requires. Specific folder structures required by the function app depend on language:

- [C# compiled \(.csproj\)](#)
- [C# script \(.csx\)](#)
- [F# script](#)
- [Java](#)
- [JavaScript](#)
- [PowerShell](#)
- [Python](#)

In version 2.x and higher of the Functions runtime, all functions in the function app must share the same language stack.

A function app includes all of the files and folders in the `wwwroot` directory. A .zip file deployment includes the

contents of the `wwwroot` directory, but not the directory itself. When deploying a C# class library project, you must include the compiled library files and dependencies in a `bin` subfolder in your .zip package.

When you are developing on a local computer, you can manually create a .zip file of the function app project folder using built-in .zip compression functionality or third-party tools.

Deploy by using Azure CLI

You can use Azure CLI to trigger a push deployment. Push deploy a .zip file to your function app by using the [az functionapp deployment source config-zip](#) command. To use this command, you must use Azure CLI version 2.0.21 or later. To see what Azure CLI version you are using, use the `az --version` command.

In the following command, replace the `<zip_file_path>` placeholder with the path to the location of your .zip file. Also, replace `<app_name>` with the unique name of your function app and replace `<resource_group>` with the name of your resource group.

```
az functionapp deployment source config-zip -g <resource_group> -n \  
<app_name> --src <zip_file_path>
```

This command deploys project files from the downloaded .zip file to your function app in Azure. It then restarts the app. To view the list of deployments for this function app, you must use the REST APIs.

When you're using Azure CLI on your local computer, `<zip_file_path>` is the path to the .zip file on your computer. You can also run Azure CLI in [Azure Cloud Shell](#). When you use Cloud Shell, you must first upload your deployment .zip file to the Azure Files account that's associated with your Cloud Shell. In that case, `<zip_file_path>` is the storage location that your Cloud Shell account uses. For more information, see [Persist files in Azure Cloud Shell](#).

Deploy ZIP file with REST APIs

You can use the [deployment service REST APIs](#) to deploy the .zip file to your app in Azure. To deploy, send a POST request to `https://<app_name>.scm.azurewebsites.net/api/zipdeploy`. The POST request must contain the .zip file in the message body. The deployment credentials for your app are provided in the request by using HTTP BASIC authentication. For more information, see the [.zip push deployment reference](#).

For the HTTP BASIC authentication, you need your App Service deployment credentials. To see how to set your deployment credentials, see [Set and reset user-level credentials](#).

With cURL

The following example uses the cURL tool to deploy a .zip file. Replace the placeholders `<deployment_user>`, `<zip_file_path>`, and `<app_name>`. When prompted by cURL, type in the password.

```
curl -X POST -u <deployment_user> --data-binary @"<zip_file_path>"  
https://<app_name>.scm.azurewebsites.net/api/zipdeploy
```

This request triggers push deployment from the uploaded .zip file. You can review the current and past deployments by using the `https://<app_name>.scm.azurewebsites.net/api/deployments` endpoint, as shown in the following cURL example. Again, replace `<app_name>` with the name of your app and `<deployment_user>` with the username of your deployment credentials.

```
curl -u <deployment_user> https://<app_name>.scm.azurewebsites.net/api/deployments
```

Asynchronous zip deployment

While deploying synchronously you may receive errors related to connection timeouts. Add `?isAsync=true` to the URL to deploy asynchronously. You will receive a response as soon as the zip file is uploaded with a `Location` header pointing to the pollable deployment status URL. When polling the URL provided in the `Location` header, you will receive a HTTP 202 (Accepted) response while the process is ongoing and a HTTP 200 (OK) response once the archive has been expanded and the deployment has completed successfully.

Azure AD authentication

An alternative to using HTTP BASIC authentication for the zip deployment is to use an Azure AD identity. Azure AD identity may be needed if [HTTP BASIC authentication is disabled for the SCM site](#).

A valid Azure AD access token for the user or service principal performing the deployment will be required. An access token can be retrieved using the Azure CLI's `az account get-access-token` command. The access token will be used in the Authentication header of the HTTP POST request.

```
curl -X POST \
--data-binary @"<zip_file_path>" \
-H "Authorization: Bearer <access_token>" \
"https://<app_name>.scm.azurewebsites.net/api/zipdeploy"
```

With PowerShell

The following example uses [Publish-AzWebapp](#) upload the .zip file. Replace the placeholders `<group-name>`, `<app-name>`, and `<zip-file-path>`.

```
Publish-AzWebapp -ResourceGroupName <group-name> -Name <app-name> -ArchivePath <zip-file-path>
```

This request triggers push deployment from the uploaded .zip file.

To review the current and past deployments, run the following commands. Again, replace the `<deployment-user>`, `<deployment-password>`, and `<app-name>` placeholders.

```
$username = "<deployment-user>"
$password = "<deployment-password>"
$apiUrl = "https://<app-name>.scm.azurewebsites.net/api/deployments"
$base64AuthInfo = [Convert]::ToBase64String([Text.Encoding]::ASCII.GetBytes("{0}:{1}" -f $username, $password))
$userAgent = "powershell/1.0"
Invoke-RestMethod -Uri $apiUrl -Headers @{Authorization=("Basic {0}" -f $base64AuthInfo)} -UserAgent $userAgent -Method GET
```

Run functions from the deployment package

You can also choose to run your functions directly from the deployment package file. This method skips the deployment step of copying files from the package to the `wwwroot` directory of your function app. Instead, the package file is mounted by the Functions runtime, and the contents of the `wwwroot` directory become read-only.

Zip deployment integrates with this feature, which you can enable by setting the function app setting `WEBSITE_RUN_FROM_PACKAGE` to a value of `1`. For more information, see [Run your functions from a deployment package file](#).

Deployment customization

The deployment process assumes that the .zip file that you push contains a ready-to-run app. By default, no customizations are run. To enable the same build processes that you get with continuous integration, add the following to your application settings:

SCM_DO_BUILD_DURING_DEPLOYMENT=true

When you use .zip push deployment, this setting is **false** by default. The default is **true** for continuous integration deployments. When set to **true**, your deployment-related settings are used during deployment. You can configure these settings either as app settings or in a .deployment configuration file that's located in the root of your .zip file. For more information, see [Repository and deployment-related settings](#) in the deployment reference.

Download your function app files

If you created your functions by using the editor in the Azure portal, you can download your existing function app project as a .zip file in one of these ways:

- **From the Azure portal:**

1. Sign in to the [Azure portal](#), and then go to your function app.
2. On the **Overview** tab, select **Download app content**. Select your download options, and then select **Download**.

The screenshot shows the Azure portal's Overview tab for a function app. At the top, there are several buttons: Stop, Swap, Restart, Download publish profile, Reset publish credentials, Download app content (which is highlighted with a red box), and Delete. Below these buttons, there are sections for Status, Subscription, Resource group, URL, and App Service plan / pricing tier. Under the URL section, the URL is listed as <https://functions-ggailey777.azurewebsites.net>. In the bottom section, there is a heading 'Configured features' followed by links to 'Function app settings' and 'Application settings'.

The downloaded .zip file is in the correct format to be republished to your function app by using .zip push deployment. The portal download can also add the files needed to open your function app directly in Visual Studio.

- **Using REST APIs:**

Use the following deployment GET API to download the files from your <function_app> project:

```
https://<function_app>.scm.azurewebsites.net/api/zip/site/wwwroot/
```

Including `/site/wwwroot/` makes sure your zip file includes only the function app project files and not the entire site. If you are not already signed in to Azure, you will be asked to do so.

You can also download a .zip file from a GitHub repository. When you download a GitHub repository as a .zip file, GitHub adds an extra folder level for the branch. This extra folder level means that you can't deploy the .zip file directly as you downloaded it from GitHub. If you're using a GitHub repository to maintain your function app, you should use [continuous integration](#) to deploy your app.

Next steps

[Run your functions from a package file in Azure](#)

Run your functions from a package file in Azure

10/5/2022 • 6 minutes to read • [Edit Online](#)

In Azure, you can run your functions directly from a deployment package file in your function app. The other option is to deploy your files in the `d:\home\site\wwwroot` (Windows) or `/home/site/wwwroot` (Linux) directory of your function app.

This article describes the benefits of running your functions from a package. It also shows how to enable this functionality in your function app.

Benefits of running from a package file

There are several benefits to running from a package file:

- Reduces the risk of file copy locking issues.
- Can be deployed to a production app (with restart).
- You can be certain of the files that are running in your app.
- Improves the performance of [Azure Resource Manager deployments](#).
- May reduce cold-start times, particularly for JavaScript functions with large npm package trees.

For more information, see [this announcement](#).

Enable functions to run from a package

To enable your function app to run from a package, add a `WEBSITE_RUN_FROM_PACKAGE` setting to your function app settings. The `WEBSITE_RUN_FROM_PACKAGE` setting can have one of the following values:

VALUE	DESCRIPTION
<code>1</code>	Indicates that the function app runs from a local package file deployed in the <code>d:\home\data\SitePackages</code> (Windows) or <code>/home/data/SitePackages</code> (Linux) folder of your function app.
<code><URL></code>	Sets a URL that is the remote location of the specific package file you want to run. Required for functions apps running on Linux in a Consumption plan.

The following table indicates the recommended `WEBSITE_RUN_FROM_PACKAGE` options for deployment to a specific operating system and hosting plan:

HOSTING PLAN	WINDOWS	LINUX
Consumption	<code>1</code> is highly recommended.	Only <code><URL></code> is supported.
Premium	<code>1</code> is recommended.	<code>1</code> is recommended.
Dedicated	<code>1</code> is recommended.	<code>1</code> is recommended.

General considerations

- The package file must be .zip formatted. Tar and gzip formats aren't currently supported.
- [Zip deployment](#) is recommended.
- When deploying your function app to Windows, you should set `WEBSITE_RUN_FROM_PACKAGE` to `1` and publish with zip deployment.
- When you run from a package, the `wwwroot` folder becomes read-only and you'll receive an error when writing files to this directory. Files are also read-only in the Azure portal.
- The maximum size for a deployment package file is currently 1 GB.
- You can't use local cache when running from a deployment package.
- If your project needs to use remote build, don't use the `WEBSITE_RUN_FROM_PACKAGE` app setting. Instead add the `SCM_DO_BUILD_DURING_DEPLOYMENT=true` deployment customization app setting. For Linux, also add the `ENABLE_ORYX_BUILD=true` setting. To learn more, see [Remote build](#).

Adding the WEBSITE_RUN_FROM_PACKAGE setting

There are several ways that you can add, update, and delete function app settings:

- [In the Azure portal](#).
- [By using the Azure CLI](#).
- [By using Azure PowerShell](#).

Changes to function app settings require your function app to be restarted.

Using WEBSITE_RUN_FROM_PACKAGE = 1

This section provides information about how to run your function app from a local package file.

Considerations for deploying from an on-site package

- Using an on-site package is the recommended option for running from the deployment package, except on Linux hosted in a Consumption plan.
- [Zip deployment](#) is the recommended way to upload a deployment package to your site.
- When not using zip deployment, make sure the `d:\home\data\SitePackages` (Windows) or `/home/data/SitePackages` (Linux) folder has a file named `packagename.txt`. This file contains only the name, without any whitespace, of the package file in this folder that's currently running.

Integration with zip deployment

[Zip deployment](#) is a feature of Azure App Service that lets you deploy your function app project to the `wwwroot` directory. The project is packaged as a .zip deployment file. The same APIs can be used to deploy your package to the `d:\home\data\SitePackages` (Windows) or `/home/data/SitePackages` (Linux) folder.

With the `WEBSITE_RUN_FROM_PACKAGE` app setting value of `1`, the zip deployment APIs copy your package to the `d:\home\data\SitePackages` (Windows) or `/home/data/SitePackages` (Linux) folder instead of extracting the files to `d:\home\site\wwwroot` (Windows) or `/home/site/wwwroot` (Linux). It also creates the `packagename.txt` file. After a restart, the package is mounted to `wwwroot` as a read-only filesystem. For more information about zip deployment, see [Zip deployment for Azure Functions](#).

NOTE

When a deployment occurs, a restart of the function app is triggered. Function executions currently running during the deploy are terminated. Please review [Improve the performance and reliability of Azure Functions](#) to learn how to write stateless and defensive functions.

Using WEBSITE_RUN_FROM_PACKAGE = URL

This section provides information about how to run your function app from a package deployed to a URL endpoint. This option is the only one supported for running from a package on Linux hosted in a Consumption plan.

Considerations for deploying from a URL

- When running a function app on Windows, the app setting `WEBSITE_RUN_FROM_PACKAGE = <URL>` gives worse cold-start performance and isn't recommended.
- When you specify a URL, you must also [manually sync triggers](#) after you publish an updated package.
- The Functions runtime must have permissions to access the package URL.
- You shouldn't deploy your package to Azure Blob Storage as a public blob. Instead, use a private container with a [Shared Access Signature \(SAS\)](#) or [use a managed identity](#) to enable the Functions runtime to access the package.
- When running on a Premium plan, make sure to [eliminate cold starts](#).
- When running on a Dedicated plan, make sure you've enabled [Always On](#).
- You can use the [Azure Storage Explorer](#) to upload package files to blob containers in your storage account.

Manually uploading a package to Blob Storage

To deploy a zipped package when using the URL option, you must create a .zip compressed deployment package and upload it to the destination. This example deploys to a container in Blob Storage.

1. Create a .zip package for your project using the utility of your choice.
2. In the [Azure portal](#), search for your storage account name or browse for it in storage accounts.
3. In the storage account, select **Containers** under **Data storage**.
4. Select **+ Container** to create a new Blob Storage container in your account.
5. In the **New container** page, provide a **Name** (for example, "deployments"), make sure the **Public access level** is **Private**, and select **Create**.
6. Select the container you created, select **Upload**, browse to the location of the .zip file you created with your project, and select **Upload**.
7. After the upload completes, choose your uploaded blob file, and copy the URL. You may need to generate a SAS URL if you aren't [using an identity](#)
8. Search for your function app or browse for it in the **Function App** page.
9. In your function app, select **Configurations** under **Settings**.
10. In the **Application Settings** tab, select **New application setting**
11. Enter the value `WEBSITE_RUN_FROM_PACKAGE` for the **Name**, and paste the URL of your package in Blob Storage as the **Value**.
12. Select **OK**. Then select **Save > Continue** to save the setting and restart the app.

Now you can run your function in Azure to verify that deployment has succeeded using the deployment package .zip file.

The following shows a function app configured to run from a .zip file hosted in Azure Blob storage:

Application settings

APP SETTING NAME	VALUE
WEBSITE_RUN_FROM_PACKAGE	https://myblobstorage.blob.core.windows.net/content/MyFunction...
+ Add new setting	

Fetch a package from Azure Blob Storage using a managed identity

Azure Blob Storage can be configured to [authorize requests with Azure AD](#). This means that instead of generating a SAS key with an expiration, you can instead rely on the application's [managed identity](#). By default, the app's system-assigned identity will be used. If you wish to specify a user-assigned identity, you can set the `WEBSITE_RUN_FROM_PACKAGE_BLOB_MI_RESOURCE_ID` app setting to the resource ID of that identity. The setting can also accept "SystemAssigned" as a value, although this is the same as omitting the setting altogether.

To enable the package to be fetched using the identity:

1. Ensure that the blob is [configured for private access](#).
2. Grant the identity the [Storage Blob Data Reader](#) role with scope over the package blob. See [Assign an Azure role for access to blob data](#) for details on creating the role assignment.
3. Set the `WEBSITE_RUN_FROM_PACKAGE` application setting to the blob URL of the package. This will likely be of the form "`https://{storage-account-name}.blob.core.windows.net/{container-name}/{path-to-package}`" or similar.

Next steps

[Continuous deployment for Azure Functions](#)

Zero-downtime deployment for Durable Functions

10/5/2022 • 6 minutes to read • [Edit Online](#)

The [reliable execution model](#) of Durable Functions requires that orchestrations be deterministic, which creates an additional challenge to consider when you deploy updates. When a deployment contains changes to activity function signatures or orchestrator logic, in-flight orchestration instances fail. This situation is especially a problem for instances of long-running orchestrations, which might represent hours or days of work.

To prevent these failures from happening, you have two options:

- Delay your deployment until all running orchestration instances have completed.
- Make sure that any running orchestration instances use the existing versions of your functions.

The following chart compares the three main strategies to achieve a zero-downtime deployment for Durable Functions:

STRATEGY	WHEN TO USE	PROS	CONS
Versioning	Applications that don't experience frequent breaking changes .	Simple to implement.	Increased function app size in memory and number of functions. Code duplication.
Status check with slot	A system that doesn't have long-running orchestrations lasting more than 24 hours or frequently overlapping orchestrations.	Simple code base. Doesn't require additional function app management.	Requires additional storage account or task hub management. Requires periods of time when no orchestrations are running.
Application routing	A system that doesn't have periods of time when orchestrations aren't running, such as those time periods with orchestrations that last more than 24 hours or with frequently overlapping orchestrations.	Handles new versions of systems with continually running orchestrations that have breaking changes.	Requires an intelligent application router. Could max out the number of function apps allowed by your subscription. The default is 100.

The remainder of this document describes these strategies in more detail.

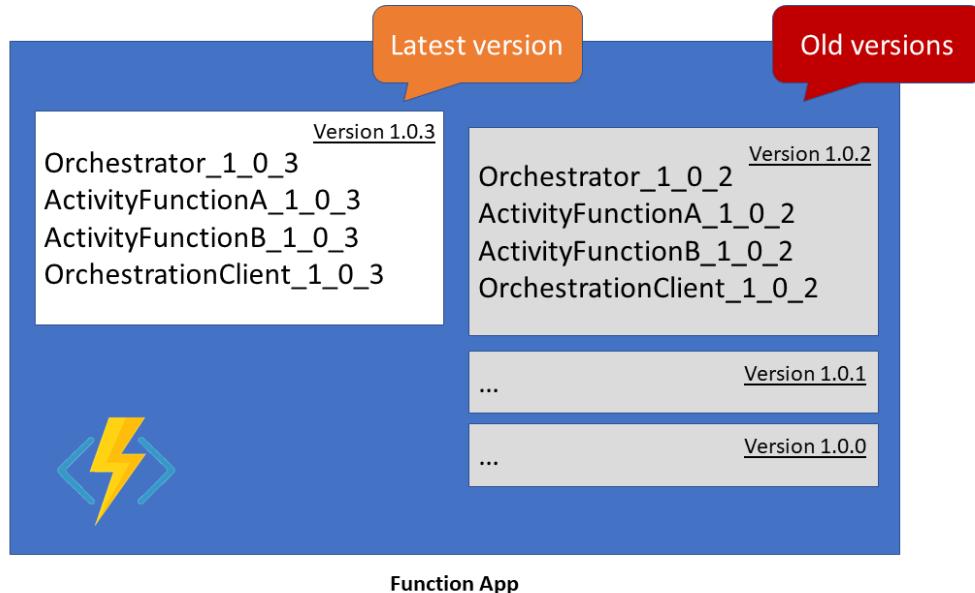
NOTE

The descriptions for these zero-downtime deployment strategies assume you are using the default Azure Storage provider for Durable Functions. The guidance may not be appropriate if you are using a storage provider other than the default Azure Storage provider. For more information on the various storage provider options and how they compare, see the [Durable Functions storage providers](#) documentation.

Versioning

Define new versions of your functions and leave the old versions in your function app. As you can see in the diagram, a function's version becomes part of its name. Because previous versions of functions are preserved, in-flight orchestration instances can continue to reference them. Meanwhile, requests for new orchestration

instances call for the latest version, which your orchestration client function can reference from an app setting.



In this strategy, every function must be copied, and its references to other functions must be updated. You can make it easier by writing a script. Here's a [sample project](#) with a migration script.

NOTE

This strategy uses deployment slots to avoid downtime during deployment. For more detailed information about how to create and use new deployment slots, see [Azure Functions deployment slots](#).

Status check with slot

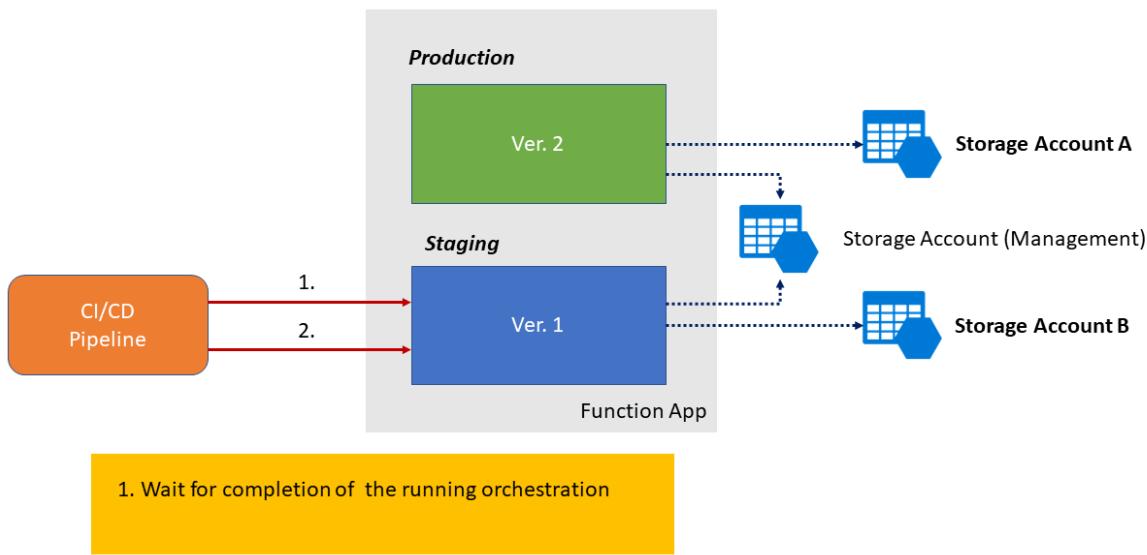
While the current version of your function app is running in your production slot, deploy the new version of your function app to your staging slot. Before you swap your production and staging slots, check to see if there are any running orchestration instances. After all orchestration instances are complete, you can do the swap. This strategy works when you have predictable periods when no orchestration instances are in flight. This is the best approach when your orchestrations aren't long-running and when your orchestration executions don't frequently overlap.

Function app configuration

Use the following procedure to set up this scenario.

1. [Add deployment slots](#) to your function app for staging and production.
2. For each slot, set the [AzureWebJobsStorage application setting](#) to the connection string of a shared storage account. This storage account connection string is used by the Azure Functions runtime to securely store the [functions' access keys](#).
3. For each slot, create a new app setting, for example, `DurableManagementStorage`. Set its value to the connection string of different storage accounts. These storage accounts are used by the Durable Functions extension for [reliable execution](#). Use a separate storage account for each slot. Don't mark this setting as a deployment slot setting.
4. In your function app's [host.json file's durableTask section](#), specify `connectionStringName` (Durable 2.x) or `azureStorageConnectionStringName` (Durable 1.x) as the name of the app setting you created in step 3.

The following diagram shows the described configuration of deployment slots and storage accounts. In this potential predeployment scenario, version 2 of a function app is running in the production slot, while version 1 remains in the staging slot.



host.json examples

The following JSON fragments are examples of the connection string setting in the *host.json* file.

Functions 2.0

```
{  
  "version": 2.0,  
  "extensions": {  
    "durableTask": {  
      "hubName": "MyTaskHub",  
      "storageProvider": {  
        "connectionStringName": "DurableManagementStorage"  
      }  
    }  
  }  
}
```

Functions 1.x

```
{  
  "durableTask": {  
    "azureStorageConnectionStringName": "DurableManagementStorage"  
  }  
}
```

CI/CD pipeline configuration

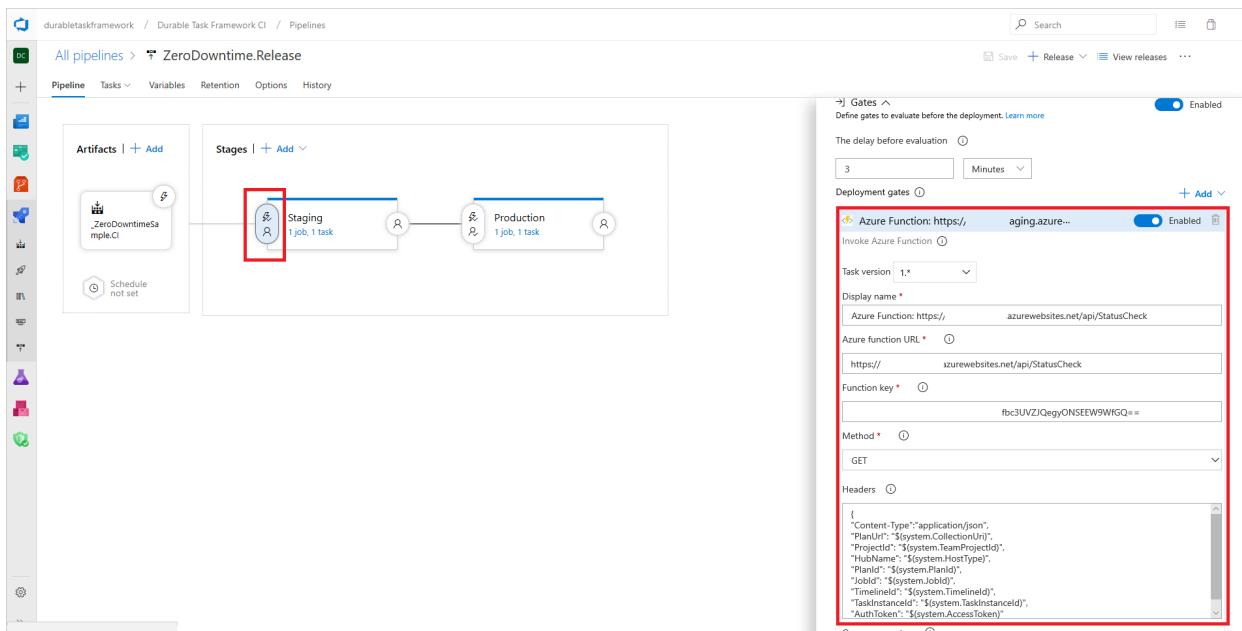
Configure your CI/CD pipeline to deploy only when your function app has no pending or running orchestration instances. When you're using Azure Pipelines, you can create a function that checks for these conditions, as in the following example:

```
[FunctionName("StatusCheck")]
public static async Task<IActionResult> StatusCheck(
    [HttpTrigger(AuthorizationLevel.Function, "get", "post")] HttpRequestMessage req,
    [DurableClient] IDurableOrchestrationClient client,
    ILogger log)
{
    var runtimeStatus = new List<OrchestrationRuntimeStatus>();

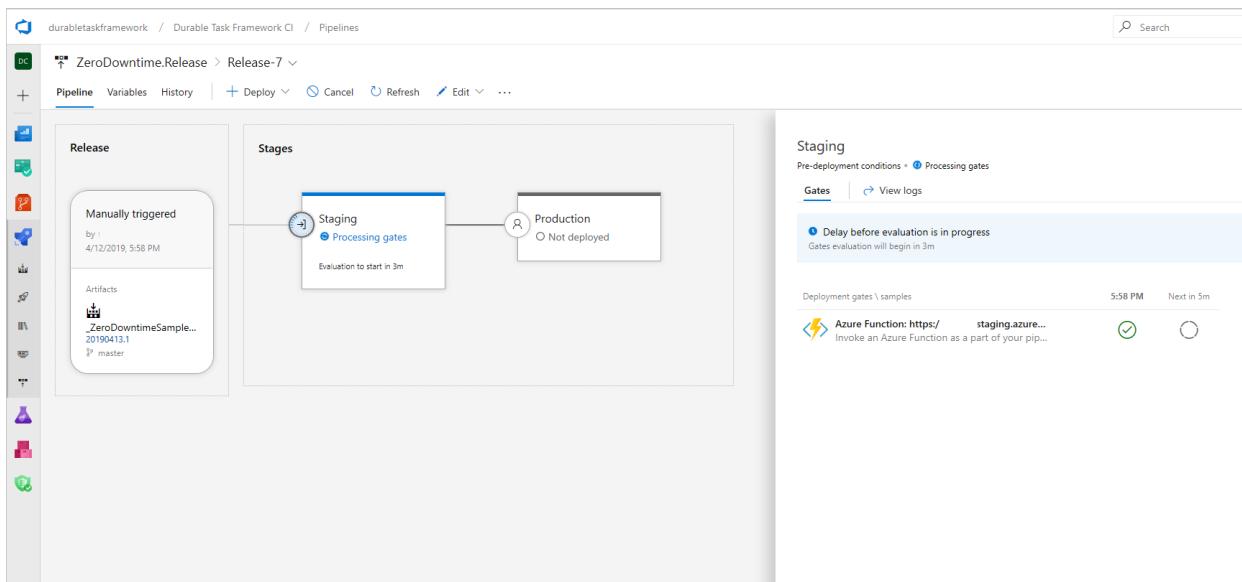
    runtimeStatus.Add(OrchestrationRuntimeStatus.Pending);
    runtimeStatus.Add(OrchestrationRuntimeStatus.Running);

    var result = await client.ListInstancesAsync(new OrchestrationStatusQueryCondition() { RuntimeStatus = runtimeStatus }, CancellationToken.None);
    return (ActionResult)new OkObjectResult(new { HasRunning = result.DurableOrchestrationState.Any() });
}
```

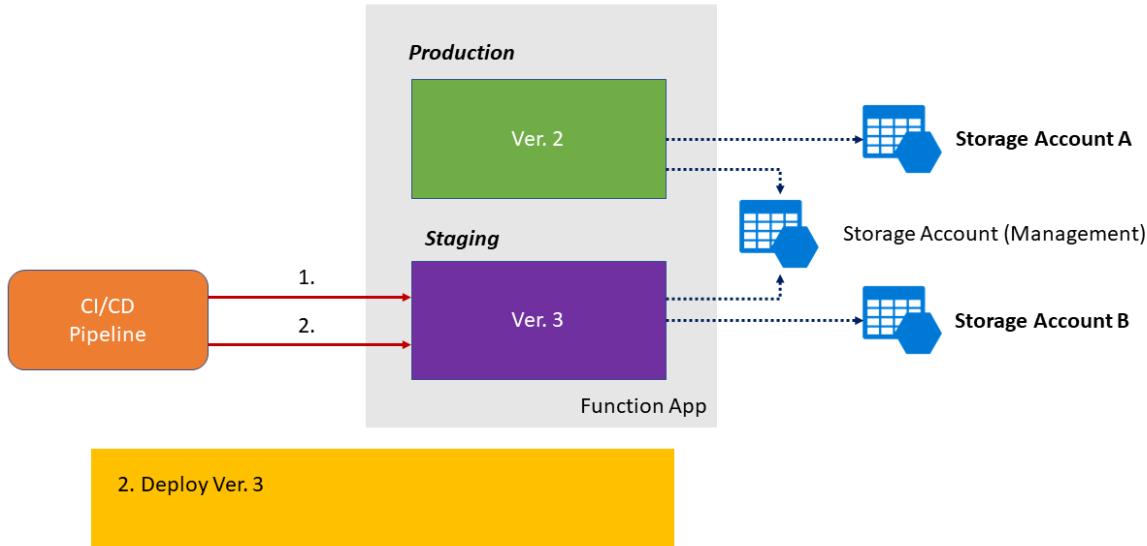
Next, configure the staging gate to wait until no orchestrations are running. For more information, see [Release deployment control using gates](#)



Azure Pipelines checks your function app for running orchestrations instances before your deployment starts.

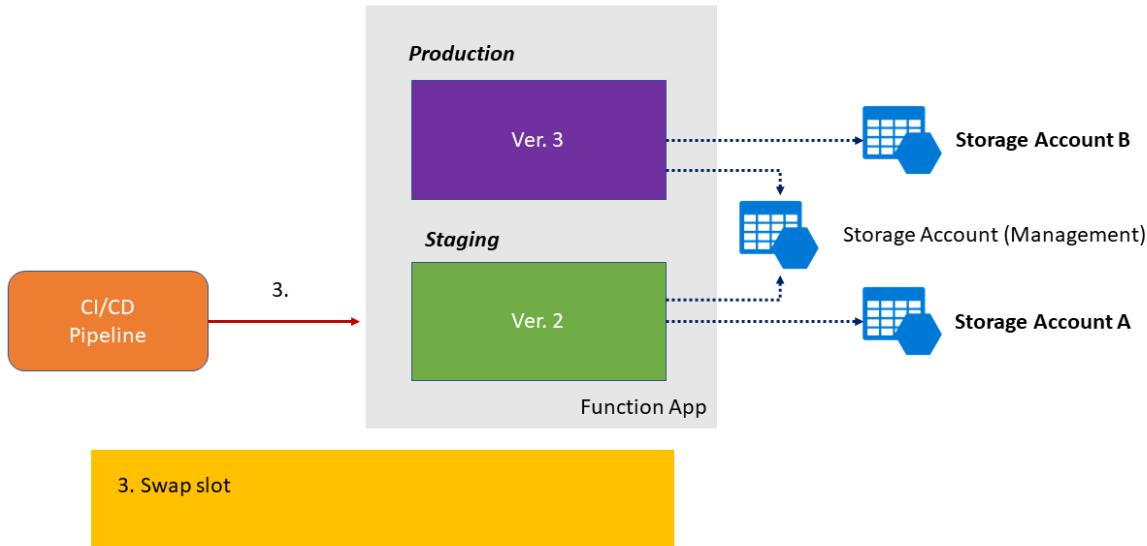


Now the new version of your function app should be deployed to the staging slot.



Finally, swap slots.

Application settings that aren't marked as deployment slot settings are also swapped, so the version 2 app keeps its reference to storage account A. Because orchestration state is tracked in the storage account, any orchestrations running on the version 2 app continue to run in the new slot without interruption.



To use the same storage account for both slots, you can change the names of your task hubs. In this case, you need to manage the state of your slots and your app's HubName settings. To learn more, see [Task hubs in Durable Functions](#).

Application routing

This strategy is the most complex. However, it can be used for function apps that don't have time between running orchestrations.

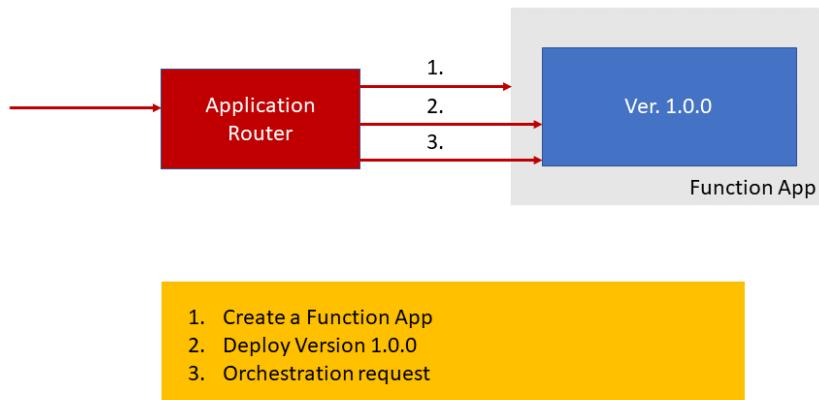
For this strategy, you must create an *application router* in front of your Durable Functions. This router can be implemented with Durable Functions. The router has the responsibility to:

- Deploy the function app.
- Manage the version of Durable Functions.
- Route orchestration requests to function apps.

The first time an orchestration request is received, the router does the following tasks:

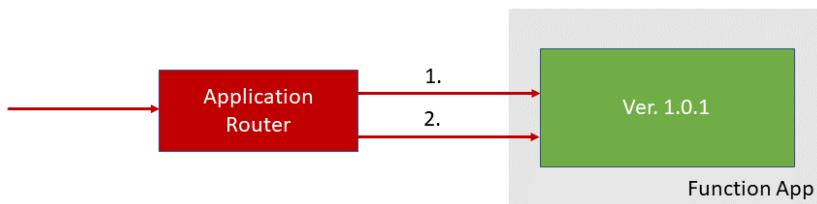
1. Creates a new function app in Azure.
2. Deploys your function app's code to the new function app in Azure.
3. Forwards the orchestration request to the new app.

The router manages the state of which version of your app's code is deployed to which function app in Azure.



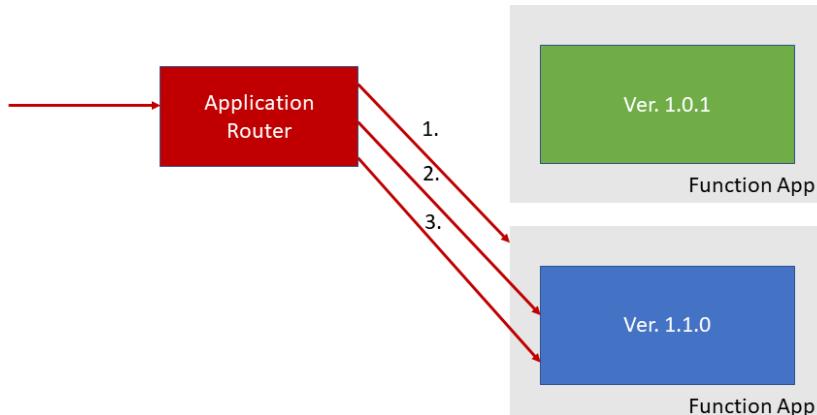
The router directs deployment and orchestration requests to the appropriate function app based on the version sent with the request. It ignores the patch version.

When you deploy a new version of your app without a breaking change, you can increment the patch version. The router deploys to your existing function app and sends requests for the old and new versions of the code, which are routed to the same function app.



- 1. Deploy Version 1.0.1
- 2. Orchestration request

When you deploy a new version of your app with a breaking change, you can increment the major or minor version. Then the application router creates a new function app in Azure, deploys to it, and routes requests for the new version of your app to it. In the following diagram, running orchestrations on the 1.0.1 version of the app keep running, but requests for the 1.1.0 version are routed to the new function app.



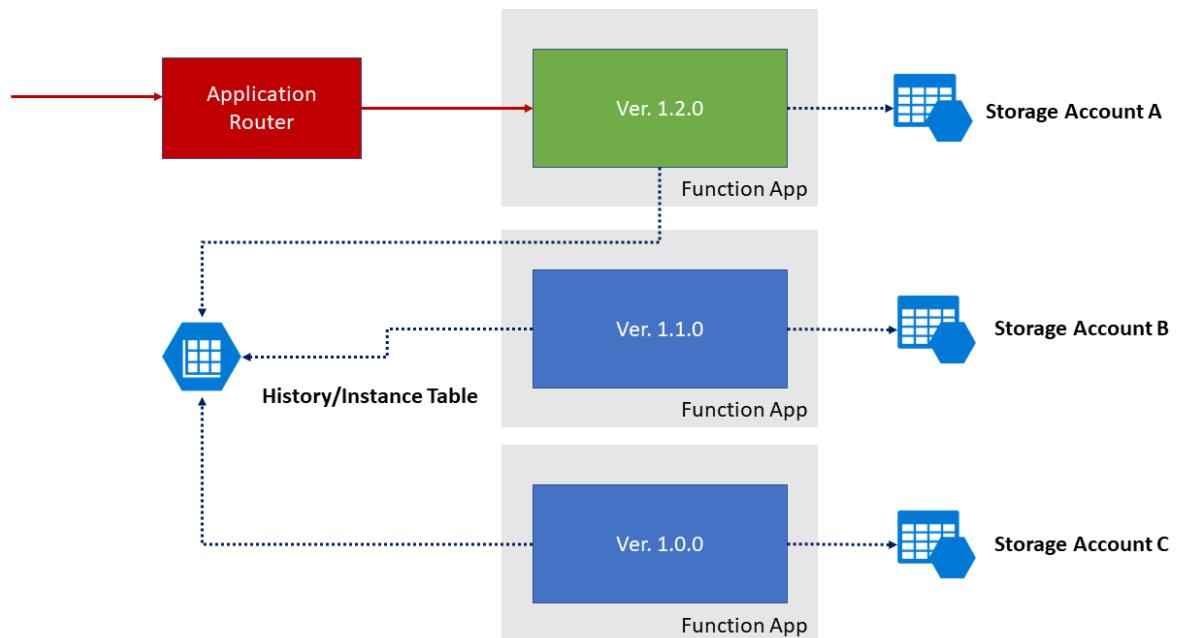
- 1. Create a Function App
- 2. Deploy Version 1.1.0
- 3. Orchestration request

The router monitors the status of orchestrations on the 1.0.1 version and removes apps after all orchestrations are finished.

Tracking store settings

Each function app should use separate scheduling queues, possibly in separate storage accounts. If you want to query all orchestrations instances across all versions of your application, you can share instance and history tables across your function apps. You can share tables by configuring the `trackingStoreConnectionStringName` and `trackingStoreNamePrefix` settings in the [host.json settings](#) file so that they all use the same values.

For more information, see [Manage instances in Durable Functions in Azure](#).



Next steps

[Versioning Durable Functions](#)

Diagnostics in Durable Functions in Azure

10/5/2022 • 13 minutes to read • [Edit Online](#)

There are several options for diagnosing issues with [Durable Functions](#). Some of these options are the same for regular functions and some of them are unique to Durable Functions.

Application Insights

[Application Insights](#) is the recommended way to do diagnostics and monitoring in Azure Functions. The same applies to Durable Functions. For an overview of how to leverage Application Insights in your function app, see [Monitor Azure Functions](#).

The Azure Functions Durable Extension also emits *tracking events* that allow you to trace the end-to-end execution of an orchestration. These tracking events can be found and queried using the [Application Insights Analytics](#) tool in the Azure portal.

Tracking data

Each lifecycle event of an orchestration instance causes a tracking event to be written to the **traces** collection in Application Insights. This event contains a **customDimensions** payload with several fields. Field names are all prepended with `prop_`.

- **hubName**: The name of the task hub in which your orchestrations are running.
- **appName**: The name of the function app. This field is useful when you have multiple function apps sharing the same Application Insights instance.
- **slotName**: The [deployment slot](#) in which the current function app is running. This field is useful when you use deployment slots to version your orchestrations.
- **functionName**: The name of the orchestrator or activity function.
- **functionType**: The type of the function, such as **Orchestrator** or **Activity**.
- **instanceId**: The unique ID of the orchestration instance.
- **state**: The lifecycle execution state of the instance. Valid values include:
 - **Scheduled**: The function was scheduled for execution but hasn't started running yet.
 - **Started**: The function has started running but has not yet awaited or completed.
 - **Awaited**: The orchestrator has scheduled some work and is waiting for it to complete.
 - **Listening**: The orchestrator is listening for an external event notification.
 - **Completed**: The function has completed successfully.
 - **Failed**: The function failed with an error.
- **reason**: Additional data associated with the tracking event. For example, if an instance is waiting for an external event notification, this field indicates the name of the event it is waiting for. If a function has failed, this field will contain the error details.
- **isReplay**: Boolean value indicating whether the tracking event is for replayed execution.
- **extensionVersion**: The version of the Durable Task extension. The version information is especially important data when reporting possible bugs in the extension. Long-running instances may report multiple versions if an update occurs while it is running.
- **sequenceNumber**: Execution sequence number for an event. Combined with the timestamp helps to order the events by execution time. *Note that this number will be reset to zero if the host restarts while the instance is running, so it's important to always sort by timestamp first, then sequenceNumber.*

The verbosity of tracking data emitted to Application Insights can be configured in the `logger` (Functions 1.x) or

`logging` (Functions 2.0) section of the `host.json` file.

Functions 1.0

```
{  
    "logger": {  
        "categoryFilter": {  
            "categoryLevels": {  
                "Host.Triggers.DurableTask": "Information"  
            }  
        }  
    }  
}
```

Functions 2.0

```
{  
    "logging": {  
        "logLevel": {  
            "Host.Triggers.DurableTask": "Information",  
        },  
    }  
}
```

By default, all *non-replay* tracking events are emitted. The volume of data can be reduced by setting `Host.Triggers.DurableTask` to `"Warning"` or `"Error"` in which case tracking events will only be emitted for exceptional situations. To enable emitting the verbose orchestration replay events, set the `logReplayEvents` to `true` in the `host.json` configuration file.

NOTE

By default, Application Insights telemetry is sampled by the Azure Functions runtime to avoid emitting data too frequently. This can cause tracking information to be lost when many lifecycle events occur in a short period of time. The [Azure Functions Monitoring article](#) explains how to configure this behavior.

Inputs and outputs of orchestrator, activity, and entity functions are not logged by default. This default behavior is recommended because logging inputs and outputs could increase Application Insights costs. Function input and output payloads may also contain sensitive information. Instead, the number of bytes for function inputs and outputs are logged instead of the actual payloads by default. If you want the Durable Functions extension to log the full input and output payloads, set the `traceInputsAndOutputs` property to `true` in the `host.json` configuration file.

Single instance query

The following query shows historical tracking data for a single instance of the [Hello Sequence](#) function orchestration. It's written using the [Kusto Query Language](#). It filters out replay execution so that only the *logical* execution path is shown. Events can be ordered by sorting by `timestamp` and `sequenceNumber` as shown in the query below:

```

let targetInstanceId = "ddd1aaa685034059b545eb004b15d4eb";
let start = datetime(2018-03-25T09:20:00);
traces
| where timestamp > start and timestamp < start + 30m
| where customDimensions.Category == "Host.Triggers.DurableTask"
| extend functionName = customDimensions["prop__functionName"]
| extend instanceId = customDimensions["prop__instanceId"]
| extend state = customDimensions["prop__state"]
| extend isReplay = tobool(tolower(customDimensions["prop__isReplay"]))
| extend sequenceNumber = tolong(customDimensions["prop__sequenceNumber"])
| where isReplay != true
| where instanceId == targetInstanceId
| sort by timestamp asc, sequenceNumber asc
| project timestamp, functionName, state, instanceId, sequenceNumber, appName = cloud_RoleName

```

The result is a list of tracking events that shows the execution path of the orchestration, including any activity functions ordered by the execution time in ascending order.

timestamp [UTC]	functionName	state	instanceId	sequenceNumber
> 2018-03-25T09:31:31.709	E1_HelloSequence	Scheduled	ddd1aaa685034059b545eb004b15d4eb	154
> 2018-03-25T09:31:31.790	E1_HelloSequence	Started	ddd1aaa685034059b545eb004b15d4eb	156
> 2018-03-25T09:31:31.820	E1_SayHello	Scheduled	ddd1aaa685034059b545eb004b15d4eb	157
> 2018-03-25T09:31:31.922	E1_HelloSequence	Awaited	ddd1aaa685034059b545eb004b15d4eb	160
> 2018-03-25T09:31:32.179	E1_SayHello	Started	ddd1aaa685034059b545eb004b15d4eb	166
> 2018-03-25T09:31:32.375	E1_SayHello	Completed	ddd1aaa685034059b545eb004b15d4eb	167
> 2018-03-25T09:31:32.675	E1_SayHello	Scheduled	ddd1aaa685034059b545eb004b15d4eb	170
> 2018-03-25T09:31:32.836	E1_HelloSequence	Awaited	ddd1aaa685034059b545eb004b15d4eb	171
> 2018-03-25T09:31:33.022	E1_SayHello	Started	ddd1aaa685034059b545eb004b15d4eb	172
> 2018-03-25T09:31:33.103	E1_SayHello	Completed	ddd1aaa685034059b545eb004b15d4eb	173
> 2018-03-25T09:31:33.321	E1_SayHello	Scheduled	ddd1aaa685034059b545eb004b15d4eb	178
> 2018-03-25T09:31:33.419	E1_HelloSequence	Awaited	ddd1aaa685034059b545eb004b15d4eb	179
> 2018-03-25T09:31:33.583	E1_SayHello	Started	ddd1aaa685034059b545eb004b15d4eb	180
> 2018-03-25T09:31:33.680	E1_SayHello	Completed	ddd1aaa685034059b545eb004b15d4eb	181
> 2018-03-25T09:31:34.057	E1_HelloSequence	Completed	ddd1aaa685034059b545eb004b15d4eb	188

Instance summary query

The following query displays the status of all orchestration instances that were run in a specified time range.

```

let start = datetime(2017-09-30T04:30:00);
traces
| where timestamp > start and timestamp < start + 1h
| where customDimensions.Category == "Host.Triggers.DurableTask"
| extend functionName = tostring(customDimensions["prop__functionName"])
| extend instanceId = tostring(customDimensions["prop__instanceId"])
| extend state = tostring(customDimensions["prop__state"])
| extend isReplay = tobool(tolower(customDimensions["prop__isReplay"]))
| extend output = tostring(customDimensions["prop__output"])
| where isReplay != true
| summarize arg_max(timestamp, *) by instanceId
| project timestamp, instanceId, functionName, state, output, appName = cloud_RoleName
| order by timestamp asc

```

The result is a list of instance IDs and their current runtime status.

timestamp [UTC]	instanceId	functionName	state	output	appName
2017-09-30T04:59:30.146	902deae5c5384289ada7cb17ae25e280	E1_HelloSequence	Completed	(49 bytes)	durablefunctions-sample-compiled
2017-09-30T05:00:00.737	fbb064e035d6444b882160870df98b57	E2_BackupSiteContent	Failed		durablefunctions-sample-compiled
2017-09-30T05:00:30.250	0d24c02c0e16443ead86a911ff113c27	E1_HelloSequence	Completed	(49 bytes)	durablefunctions-sample-compiled
2017-09-30T05:00:30.310	254a8ca78b7049d5acb62fa760a71b03	E1_HelloSequence	Completed	(49 bytes)	durablefunctions-sample-compiled
2017-09-30T05:01:14.878	4652f7a63d024ddc9b95ded383557bfd	E3_Counter	Listening		durablefunctions-sample-compiled
2017-09-30T05:02:09.318	dab5b982e89d47c1aaeec54880df10b4	E4_SmsPhoneVerification	Failed		durablefunctions-sample-compiled
2017-09-30T05:02:38.120	86b51576ef53459b9b3aca0e5b507b0a	E1_HelloSequence	Completed	(49 bytes)	durablefunctions-sample-compiled
2017-09-30T05:04:48.144	6395013d2ecf4ff18433c9745f7731a7	E3_Counter	Started		durablefunctions-sample-compiled

Durable Task Framework Logging

The Durable extension logs are useful for understanding the behavior of your orchestration logic. However, these logs don't always contain enough information to debug framework-level performance and reliability issues. Starting in v2.3.0 of the Durable extension, logs emitted by the underlying Durable Task Framework (DTFx) are also available for collection.

When looking at logs emitted by the DTFx, it's important to understand that the DTFx engine is composed of two components: the core dispatch engine (`DurableTask.Core`) and one of many supported storage providers (Durable Functions uses `DurableTask.AzureStorage` by default but [other options are available](#)).

- **DurableTask.Core**: Core orchestration execution and low-level scheduling logs and telemetry.
- **DurableTask.AzureStorage**: Backend logs specific to the Azure Storage state provider. These logs include detailed interactions with the internal queues, blobs, and storage tables used to store and fetch internal orchestration state.
- **DurableTask.Netherite**: Backend logs specific to the [Netherite storage provider](#), if enabled.
- **DurableTask.SqlServer**: Backend logs specific to the [Microsoft SQL \(MSSQL\) storage provider](#), if enabled.

You can enable these logs by updating the `logging/logLevel` section of your function app's `host.json` file. The following example shows how to enable warning and error logs from both `DurableTask.Core` and `DurableTask.AzureStorage`:

```
{
  "version": "2.0",
  "logging": {
    "logLevel": {
      "DurableTask.AzureStorage": "Warning",
      "DurableTask.Core": "Warning"
    }
  }
}
```

If you have Application Insights enabled, these logs will be automatically added to the `trace` collection. You can search them the same way that you search for other `trace` logs using Kusto queries.

NOTE

For production applications, it is recommended that you enable `DurableTask.Core` and the appropriate storage provider (e.g. `DurableTask.AzureStorage`) logs using the `"Warning"` filter. Higher verbosity filters such as `"Information"` are very useful for debugging performance issues. However, these log events can be high-volume and can significantly increase Application Insights data storage costs.

The following Kusto query shows how to query for DTFx logs. The most important part of the query is `where customerDimensions.Category startswith "DurableTask"` since that filters the results to logs in the `DurableTask.Core` and `DurableTask.AzureStorage` categories.

```

traces
| where customDimensions.Category startswith "DurableTask"
| project
    timestamp,
    severityLevel,
    Category = customDimensions.Category,
    EventId = customDimensions.EventId,
    message,
    customDimensions
| order by timestamp asc

```

The result is a set of logs written by the Durable Task Framework log providers.

timestamp [UTC]	severityLevel	Category	EventId	message
> 8/13/2020, 12:52:36.250 AM	1	DurableTask.Core	11	Durable task hub worker started successfully after 2632ms
> 8/13/2020, 12:53:44.843 AM	1	DurableTask.AzureStorage	101	Sending [EventRaised] message to samplehubvs-control-00 for instance '@counter@20200813-125344-0000000000000000'
> 8/13/2020, 12:53:45.016 AM	1	DurableTask.AzureStorage	102	@counter@20200813-125344-0000000000000000: Fetched [EventRaised] message from samplehubvs-control-00 (delay = 1014ms)
> 8/13/2020, 12:53:45.135 AM	1	DurableTask.AzureStorage	117	samplehubvs-control-00: No new messages were found - backing off
> 8/13/2020, 12:53:45.206 AM	1	DurableTask.AzureStorage	110	@counter@20200813-125344-0000000000000000: No history events were found
> 8/13/2020, 12:53:45.218 AM	1	DurableTask.AzureStorage	140	@counter@20200813-125344-0000000000000000: Processing [EventRaised] (total delay = 1217ms)
> 8/13/2020, 12:53:45.238 AM	1	DurableTask.Core	51	@counter@20200813-125344-0000000000000000: Executing '@counter@20200813-125344-0000000000000000' orchestration logic
> 8/13/2020, 12:53:45.376 AM	1	DurableTask.Core	52	@counter@20200813-125344-0000000000000000: Orchestration '@counter@20200813-125344-0000000000000000' awaited and scheduled 1 durable operation(s).
> 8/13/2020, 12:53:45.386 AM	1	DurableTask.Core	49	@counter@20200813-125344-0000000000000000: Orchestration completed with a 'ContinuedAsNew' status and 51 bytes of output. Details:
> 8/13/2020, 12:53:45.401 AM	1	DurableTask.Core	51	@counter@20200813-125344-0000000000000000: Executing '@counter@20200813-125344-0000000000000000' orchestration logic
> 8/13/2020, 12:53:45.411 AM	1	DurableTask.Core	52	@counter@20200813-125344-0000000000000000: Orchestration '@counter@20200813-125344-0000000000000000' awaited and scheduled 0 durable operation(s).
> 8/13/2020, 12:53:45.654 AM	1	DurableTask.AzureStorage	111	@counter@20200813-125344-0000000000000000: Appended 3 new events to the history table in 184ms
> 8/13/2020, 12:53:45.727 AM	1	DurableTask.AzureStorage	135	@counter@20200813-125344-0000000000000000: Updated Instances table and set the runtime status to 'Running'
> 8/13/2020, 12:53:45.743 AM	1	DurableTask.AzureStorage	103	@counter@20200813-125344-0000000000000000: Deleting [EventRaised] message from samplehubvs-control-00

For more information about what log events are available, see the [Durable Task Framework structured logging documentation on GitHub](#).

App Logging

It's important to keep the orchestrator replay behavior in mind when writing logs directly from an orchestrator function. For example, consider the following orchestrator function:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

```

[FunctionName("FunctionChain")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context,
    ILogger log)
{
    log.LogInformation("Calling F1.");
    await context.CallActivityAsync("F1");
    log.LogInformation("Calling F2.");
    await context.CallActivityAsync("F2");
    log.LogInformation("Calling F3.");
    await context.CallActivityAsync("F3");
    log.LogInformation("Done!");
}

```

The resulting log data is going to look something like the following example output:

```
Calling F1.  
Calling F1.  
Calling F2.  
Calling F1.  
Calling F2.  
Calling F3.  
Calling F1.  
Calling F2.  
Calling F3.  
Done!
```

NOTE

Remember that while the logs claim to be calling F1, F2, and F3, those functions are only *actually* called the first time they are encountered. Subsequent calls that happen during replay are skipped and the outputs are replayed to the orchestrator logic.

If you want to only write logs on non-replay executions, you can write a conditional expression to log only if the "is replaying" flag is `false`. Consider the example above, but this time with replay checks.

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

```
[FunctionName("FunctionChain")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context,
    ILogger log)
{
    if (!context.IsReplaying) log.LogInformation("Calling F1.");
    await context.CallActivityAsync("F1");
    if (!context.IsReplaying) log.LogInformation("Calling F2.");
    await context.CallActivityAsync("F2");
    if (!context.IsReplaying) log.LogInformation("Calling F3");
    await context.CallActivityAsync("F3");
    log.LogInformation("Done!");
}
```

Starting in Durable Functions 2.0, .NET orchestrator functions also have the option to create an `ILogger` that automatically filters out log statements during replay. This automatic filtering is done using the [`IDurableOrchestrationContext.CreateReplaySafeLogger\(ILogger\)`](#) API.

```
[FunctionName("FunctionChain")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context,
    ILogger log)
{
    log = context.CreateReplaySafeLogger(log);
    log.LogInformation("Calling F1.");
    await context.CallActivityAsync("F1");
    log.LogInformation("Calling F2.");
    await context.CallActivityAsync("F2");
    log.LogInformation("Calling F3");
    await context.CallActivityAsync("F3");
    log.LogInformation("Done!");
}
```

NOTE

The previous C# examples are for Durable Functions 2.x. For Durable Functions 1.x, you must use `DurableOrchestrationContext` instead of `IDurableOrchestrationContext`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

With the previously mentioned changes, the log output is as follows:

```
Calling F1.  
Calling F2.  
Calling F3.  
Done!
```

Custom Status

Custom orchestration status lets you set a custom status value for your orchestrator function. This custom status is then visible to external clients via the [HTTP status query API](#) or via language-specific API calls. The custom orchestration status enables richer monitoring for orchestrator functions. For example, the orchestrator function code can invoke the "set custom status" API to update the progress for a long-running operation. A client, such as a web page or other external system, could then periodically query the HTTP status query APIs for richer progress information. Sample code for setting a custom status value in an orchestrator function is provided below:

- [C#](#)
- [JavaScript](#)
- [Python](#)
- [Java](#)

```
[FunctionName("SetStatusTest")]
public static async Task SetStatusTest([OrchestrationTrigger] IDurableOrchestrationContext context)
{
    // ...do work...

    // update the status of the orchestration with some arbitrary data
    var customStatus = new { completionPercentage = 90.0, status = "Updating database records" };
    context.SetCustomStatus(customStatus);

    // ...do more work...
}
```

NOTE

The previous C# example is for Durable Functions 2.x. For Durable Functions 1.x, you must use `DurableOrchestrationContext` instead of `IDurableOrchestrationContext`. For more information about the differences between versions, see the [Durable Functions versions](#) article.

While the orchestration is running, external clients can fetch this custom status:

```
GET /runtime/webhooks/durabletask/instances/instance123?code=XYZ
```

Clients will get the following response:

```
{  
    "runtimeStatus": "Running",  
    "input": null,  
    "customStatus": { "completionPercentage": 90.0, "status": "Updating database records" },  
    "output": null,  
    "createdTime": "2017-10-06T18:30:24Z",  
    "lastUpdatedTime": "2017-10-06T19:40:30Z"  
}
```

WARNING

The custom status payload is limited to 16 KB of UTF-16 JSON text because it needs to be able to fit in an Azure Table Storage column. You can use external storage if you need larger payload.

Debugging

Azure Functions supports debugging function code directly, and that same support carries forward to Durable Functions, whether running in Azure or locally. However, there are a few behaviors to be aware of when debugging:

- **Replay:** Orchestrator functions regularly [replay](#) when new inputs are received. This behavior means a single *logical* execution of an orchestrator function can result in hitting the same breakpoint multiple times, especially if it is set early in the function code.
- **Await:** Whenever an `await` is encountered in an orchestrator function, it yields control back to the Durable Task Framework dispatcher. If it is the first time a particular `await` has been encountered, the associated task is *never* resumed. Because the task never resumes, stepping *over* the await (F10 in Visual Studio) is not possible. Stepping over only works when a task is being replayed.
- **Messaging timeouts:** Durable Functions internally uses queue messages to drive execution of orchestrator, activity, and entity functions. In a multi-VM environment, breaking into the debugging for extended periods of time could cause another VM to pick up the message, resulting in duplicate execution. This behavior exists for regular queue-trigger functions as well, but is important to point out in this context since the queues are an implementation detail.
- **Stopping and starting:** Messages in Durable functions persist between debug sessions. If you stop debugging and terminate the local host process while a durable function is executing, that function may re-execute automatically in a future debug session. This behavior can be confusing when not expected. Using a [fresh task hub](#) or clearing the task hub contents between debug sessions is one technique to avoid this behavior.

TIP

When setting breakpoints in orchestrator functions, if you want to only break on non-replay execution, you can set a conditional breakpoint that breaks only if the "is replaying" value is `false`.

Storage

By default, Durable Functions stores state in Azure Storage. This behavior means you can inspect the state of your orchestrations using tools such as [Microsoft Azure Storage Explorer](#).

The screenshot shows the Microsoft Azure Storage Explorer interface. In the left sidebar, under 'Storage Accs', there are several entries: 'azururfur', 'azurefur', 'azurefur', 'azurefur', 'azurefur', 'cgillums', and 'durablefun'. Under 'Que', there is a single entry: 'Que'. The main pane displays a table titled 'DurableFunctionsHubHistory'. The table has columns: RowKey, Timestamp, EventId, EventType, IsPlayed, Name, Version, OrchestrationStatus, and Input. There are 8 rows of data:

	RowKey	Timestamp	EventId	EventType	IsPlayed	Name	Version	OrchestrationStatus	Input
1	jcd38f22c 1cf7f35afc9f4c3d89385de33d7d8ca5.0000000000000000	2017-04-29T00:45:49.244Z	-1	OrchestratorStarted	false				
2	jcd38f22c 1cf7f35afc9f4c3d89385de33d7d8ca5.0000000000000001	2017-04-29T00:45:49.244Z	-1	ExecutionStarted	true	ProcessWorkBatch			0
3	jcd38f22c 1cf7f35afc9f4c3d89385de33d7d8ca5.0000000000000002	2017-04-29T00:45:49.245Z	0	TaskScheduled	false	HelloWorld			
4	jcd38f22c 1cf7f35afc9f4c3d89385de33d7d8ca5.0000000000000003	2017-04-29T00:45:49.245Z	-1	OrchestratorCompleted	false				
5	jcd38f22c 1cf7f35afc9f4c3d89385de33d7d8ca5.0000000000000004	2017-04-29T00:45:50.962Z	-1	OrchestratorStarted	false				
6	jcd38f22c 1cf7f35afc9f4c3d89385de33d7d8ca5.0000000000000005	2017-04-29T00:45:50.962Z	-1	TaskCompleted	true				
7	jcd38f22c 1cf7f35afc9f4c3d89385de33d7d8ca5.0000000000000006	2017-04-29T00:45:50.962Z	1	ExecutionCompleted	false				Completed
8	jcd38f22c 1cf7f35afc9f4c3d89385de33d7d8ca5.0000000000000007	2017-04-29T00:45:50.963Z	-1	OrchestratorCompleted	false				

Showing 1 to 8 of 8 cached items

This is useful for debugging because you see exactly what state an orchestration may be in. Messages in the queues can also be examined to learn what work is pending (or stuck in some cases).

WARNING

While it's convenient to see execution history in table storage, avoid taking any dependency on this table. It may change as the Durable Functions extension evolves.

NOTE

Other storage providers can be configured instead of the default Azure Storage provider. Depending on the storage provider configured for your app, you may need to use different tools to inspect the underlying state. For more information, see the [Durable Functions Storage Providers](#) documentation.

3rd party tools

The Durable Functions community publishes a variety of tools that can be useful for debugging, diagnostics, or monitoring. One such tool is the open source [Durable Functions Monitor](#), a graphical tool for monitoring, managing, and debugging your orchestration instances.

Next steps

[Learn more about monitoring in Azure Functions](#)

Monitor executions in Azure Functions

10/5/2022 • 7 minutes to read • [Edit Online](#)

Azure Functions offers built-in integration with [Azure Application Insights](#) to monitor functions executions. This article provides an overview of the monitoring capabilities provided by Azure for monitoring Azure Functions.

Application Insights collects log, performance, and error data. By automatically detecting performance anomalies and featuring powerful analytics tools, you can more easily diagnose issues and better understand how your functions are used. These tools are designed to help you continuously improve performance and usability of your functions. You can even use Application Insights during local function app project development. For more information, see [What is Application Insights?](#).

As Application Insights instrumentation is built into Azure Functions, you need a valid instrumentation key to connect your function app to an Application Insights resource. The instrumentation key is added to your application settings as you create your function app resource in Azure. If your function app doesn't already have this key, you can [set it manually](#).

You can also monitor the function app itself by using Azure Monitor. To learn more, see [Monitoring Azure Functions with Azure Monitor](#).

Application Insights pricing and limits

You can try out Application Insights integration with Azure Functions for free featuring a daily limit to how much data is processed for free.

If you enable Applications Insights during development, you might hit this limit during testing. Azure provides portal and email notifications when you're approaching your daily limit. If you miss those alerts and hit the limit, new logs won't appear in Application Insights queries. Be aware of the limit to avoid unnecessary troubleshooting time. For more information, see [Application Insights billing](#).

IMPORTANT

Application Insights has a [sampling](#) feature that can protect you from producing too much telemetry data on completed executions at times of peak load. Sampling is enabled by default. If you appear to be missing data, you might need to adjust the sampling settings to fit your particular monitoring scenario. To learn more, see [Configure sampling](#).

The full list of Application Insights features available to your function app is detailed in [Application Insights for Azure Functions supported features](#).

Application Insights integration

Typically, you create an Application Insights instance when you create your function app. In this case, the instrumentation key required for the integration is already set as an application setting named `APPINSIGHTS_INSTRUMENTATIONKEY`. If for some reason your function app doesn't have the instrumentation key set, you need to [enable Application Insights integration](#).

IMPORTANT

Sovereign clouds, such as Azure Government, require the use of the Application Insights connection string (`APPLICATIONINSIGHTS_CONNECTION_STRING`) instead of the instrumentation key. To learn more, see the [APPLICATIONINSIGHTS_CONNECTION_STRING reference](#).

Collecting telemetry data

With Application Insights integration enabled, telemetry data is sent to your connected Application Insights instance. This data includes logs generated by the Functions host, traces written from your functions code, and performance data.

NOTE

In addition to data from your functions and the Functions host, you can also collect data from the [Functions scale controller](#).

Log levels and categories

When you write traces from your application code, you should assign a log level to the traces. Log levels provide a way for you to limit the amount of data that is collected from your traces.

A *log level* is assigned to every log. The value is an integer that indicates relative importance:

LOGLEVEL	CODE	DESCRIPTION
Trace	0	Logs that contain the most detailed messages. These messages might contain sensitive application data. These messages are disabled by default and should never be enabled in a production environment.
Debug	1	Logs that are used for interactive investigation during development. These logs should primarily contain information useful for debugging and have no long-term value.
Information	2	Logs that track the general flow of the application. These logs should have long-term value.
Warning	3	Logs that highlight an abnormal or unexpected event in the application flow, but don't otherwise cause the application execution to stop.
Error	4	Logs that highlight when the current flow of execution is stopped because of a failure. These errors should indicate a failure in the current activity, not an application-wide failure.

LOGLEVEL	CODE	DESCRIPTION
Critical	5	Logs that describe an unrecoverable application or system crash, or a catastrophic failure that requires immediate attention.
None	6	Disables logging for the specified category.

The [host.json](#) file configuration determines how much logging a functions app sends to Application Insights.

To learn more about log levels, see [Configure log levels](#).

By assigning logged items to a category, you have more control over telemetry generated from specific sources in your function app. Categories make it easier to run analytics over collected data. Traces written from your function code are assigned to individual categories based on the function name. To learn more about categories, see [Configure categories](#).

Custom telemetry data

In [C#](#), [JavaScript](#), and [Python](#), you can use an Application Insights SDK to write custom telemetry data.

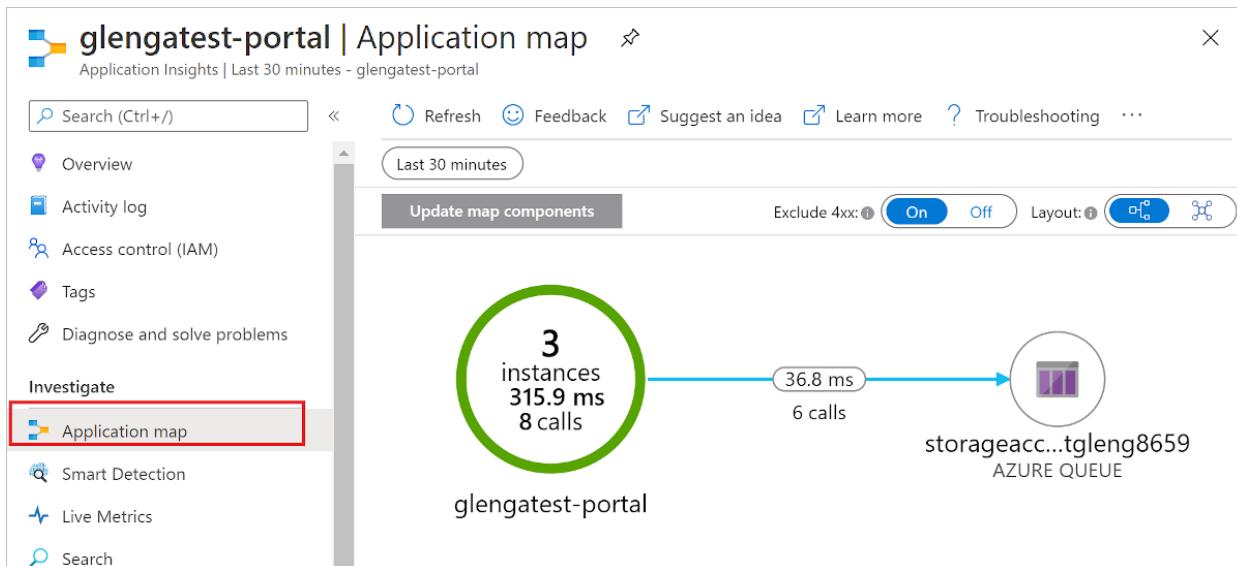
Dependencies

Starting with version 2.x of Functions, Application Insights automatically collects data on dependencies for bindings that use certain client SDKs. Application Insights distributed tracing and dependency tracking aren't currently supported for C# apps running in an [isolated process](#). Application Insights collects data on the following dependencies:

- Azure Cosmos DB
- Azure Event Hubs
- Azure Service Bus
- Azure Storage services (Blob, Queue, and Table)

HTTP requests and database calls using `SqlClient` are also captured. For the complete list of dependencies supported by Application Insights, see [automatically tracked dependencies](#).

Application Insights generates an *application map* of collected dependency data. The following is an example of an application map of an HTTP trigger function with a Queue storage output binding.



Dependencies are written at the `Information` level. If you filter at `Warning` or above, you won't see the

dependency data. Also, automatic collection of dependencies happens at a non-user scope. To capture dependency data, make sure the level is set to at least `Information` outside the user scope (`Function.<YOUR_FUNCTION_NAME>.User`) in your host.

In addition to automatic dependency data collection, you can also use one of the language-specific Application Insights SDKs to write custom dependency information to the logs. For an example how to write custom dependencies, see one of the following language-specific examples:

- [Log custom telemetry in C# functions](#)
- [Log custom telemetry in JavaScript functions](#)
- [Log custom telemetry in Python functions](#)

Writing to logs

The way that you write to logs and the APIs you use depend on the language of your function app project. See the developer guide for your language to learn more about writing logs from your functions.

- [C# \(.NET class library\)](#)
- [Java](#)
- [JavaScript](#)
- [PowerShell](#)
- [Python](#)

Analyze data

By default, the data collected from your function app is stored in Application Insights. In the [Azure portal](#), Application Insights provides an extensive set of visualizations of your telemetry data. You can drill into error logs and query events and metrics. To learn more, including basic examples of how to view and query your collected data, see [Analyze Azure Functions telemetry in Application Insights](#).

Streaming Logs

While developing an application, you often want to see what's being written to the logs in near real time when running in Azure.

There are two ways to view a stream of the log data being generated by your function executions.

- **Built-in log streaming:** the App Service platform lets you view a stream of your application log files. This stream is equivalent to the output seen when you debug your functions during [local development](#) and when you use the [Test](#) tab in the portal. All log-based information is displayed. For more information, see [Stream logs](#). This streaming method supports only a single instance, and can't be used with an app running on Linux in a Consumption plan.
- **Live Metrics Stream:** when your function app is [connected to Application Insights](#), you can view log data and other metrics in near real time in the Azure portal using [Live Metrics Stream](#). Use this method when monitoring functions running on multiple-instances or on Linux in a Consumption plan. This method uses [sampled data](#).

Log streams can be viewed both in the portal and in most local development environments. To learn how to enable log streams, see [Enable streaming execution logs in Azure Functions](#).

Diagnostic logs

Application Insights lets you export telemetry data to long-term storage or other analysis services.

Because Functions also integrates with Azure Monitor, you can also use diagnostic settings to send telemetry data to various destinations, including Azure Monitor logs. To learn more, see [Monitoring Azure Functions with Azure Monitor Logs](#).

Scale controller logs

The [Azure Functions scale controller](#) monitors instances of the Azure Functions host on which your app runs. This controller makes decisions about when to add or remove instances based on current performance. You can have the scale controller emit logs to Application Insights to better understand the decisions the scale controller is making for your function app. You can also store the generated logs in Blob storage for analysis by another service.

To enable this feature, you add an application setting named `SCALE_CONTROLLER_LOGGING_ENABLED` to your function app settings. To learn how, see [Configure scale controller logs](#).

Azure Monitor metrics

In addition to log-based telemetry data collected by Application Insights, you can also get data about how the function app is running from [Azure Monitor Metrics](#). To learn more, see [Monitoring with Azure Monitor](#).

Report issues

To report an issue with Application Insights integration in Functions, or to make a suggestion or request, [create an issue in GitHub](#).

Next steps

For more information, see the following resources:

- [Application Insights](#)
- [ASP.NET Core logging](#)

Durable Functions billing

10/5/2022 • 3 minutes to read • [Edit Online](#)

Durable Functions is billed the same way as Azure Functions. For more information, see [Azure Functions pricing](#).

When executing orchestrator functions in Azure Functions [Consumption plan](#), you need to be aware of some billing behaviors. The following sections describe these behaviors and their effect in more detail.

Orchestrator function replay billing

Orchestrator functions might replay several times throughout the lifetime of an orchestration. Each replay is viewed by the Azure Functions runtime as a distinct function invocation. For this reason, in the Azure Functions Consumption plan you're billed for each replay of an orchestrator function. Other plan types don't charge for orchestrator function replay.

Awaiting and yielding in orchestrator functions

When an orchestrator function waits for an asynchronous task to complete, the runtime considers that particular function invocation to be finished. The billing for the orchestrator function stops at that point. It doesn't resume until the next orchestrator function replay. You aren't billed for any time spent awaiting or yielding in an orchestrator function.

NOTE

Functions calling other functions is considered by some to be a Serverless anti-pattern. This is because of a problem known as *double billing*. When a function calls another function directly, both run at the same time. The called function is actively running code while the calling function is waiting for a response. In this case, you must pay for the time the calling function spends waiting for the called function to run.

There is no double billing in orchestrator functions. An orchestrator function's billing stops while it waits for the result of an activity function or sub-orchestration.

Durable HTTP polling

Orchestrator functions can make long-running HTTP calls to external endpoints as described in the [HTTP features article](#). The "call HTTP" APIs might internally poll an HTTP endpoint while following the [asynchronous 202 pattern](#).

There currently isn't direct billing for internal HTTP polling operations. However, internal polling might cause the orchestrator function to periodically replay. You'll be billed standard charges for these internal function replays.

Azure Storage transactions

Durable Functions uses Azure Storage by default to keep state persistent, process messages, and manage partitions via blob leases. Because you own this storage account, any transaction costs are billed to your Azure subscription. For more information about the Azure Storage artifacts used by Durable Functions, see the [Task hubs article](#).

Several factors contribute to the actual Azure Storage costs incurred by your Durable Functions app:

- A single function app is associated with a single task hub, which shares a set of Azure Storage resources.

These resources are used by all durable functions in a function app. The actual number of functions in the function app has no effect on Azure Storage transaction costs.

- Each function app instance internally polls multiple queues in the storage account by using an exponential-backoff polling algorithm. An idle app instance polls the queues less often than does an active app, which results in fewer transaction costs. For more information about Durable Functions queue-polling behavior when using the Azure Storage provider, see the [queue-polling section](#) of the Azure Storage provider documentation.
- When running in the Azure Functions Consumption or Premium plans, the [Azure Functions scale controller](#) regularly polls all task-hub queues in the background. If a function app is under light to moderate scale, only a single scale controller instance will poll these queues. If the function app scales out to a large number of instances, more scale controller instances might be added. These additional scale controller instances can increase the total queue-transaction costs.
- Each function app instance competes for a set of blob leases. These instances will periodically make calls to the Azure Blob service either to renew held leases or to attempt to acquire new leases. The task hub's configured partition count determines the number of blob leases. Scaling out to a larger number of function app instances likely increases the Azure Storage transaction costs associated with these lease operations.

You can find more information on Azure Storage pricing in the [Azure Storage pricing](#) documentation.

Next steps

[Learn more about Azure Functions pricing](#)