100 XP

Introduction

3 minutes

Durable Functions is an extension of Azure Functions that lets you write stateful functions in a serverless compute environment.

After completing this module, you'll be able to:

- Describe the app patterns typically used in durable functions
- Describe the four durable function types
- Explain the function Task Hubs perform in durable functions
- Describe the use of durable orchestrations, timers, and events

Next unit: Explore Durable Functions app patterns

Continue

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Explore Durable Functions app patterns

8 minutes

The *durable functions* 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 currently supports the following languages:

- **C#**: both precompiled class libraries and C# script.
- **JavaScript**: supported only for version 2.x of the Azure Functions runtime. Requires version 1.7.0 of the Durable Functions extension, or a later version.
- **Python**: requires version 2.3.1 of the Durable Functions extension, or a later version.
- **F#**: precompiled class libraries and F# script. F# script is only supported for version 1.x of the Azure Functions runtime.
- **PowerShell**: Supported only for version 3.x of the Azure Functions runtime and PowerShell7. Requires version 2.x of the bundle extensions.

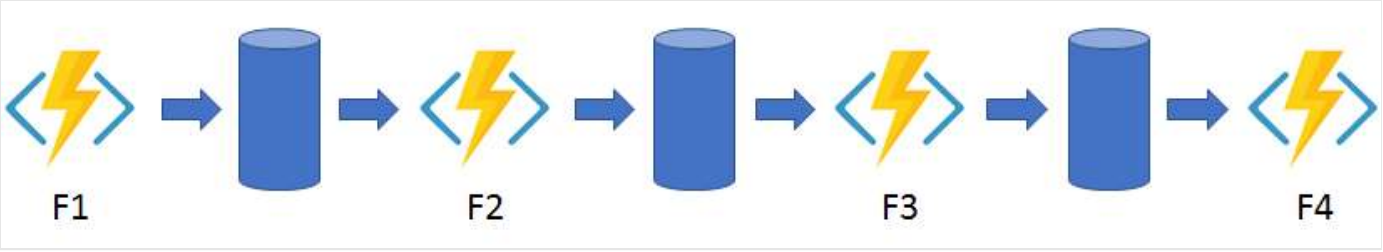
Application patterns

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
- Monitor
- Human interaction

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.



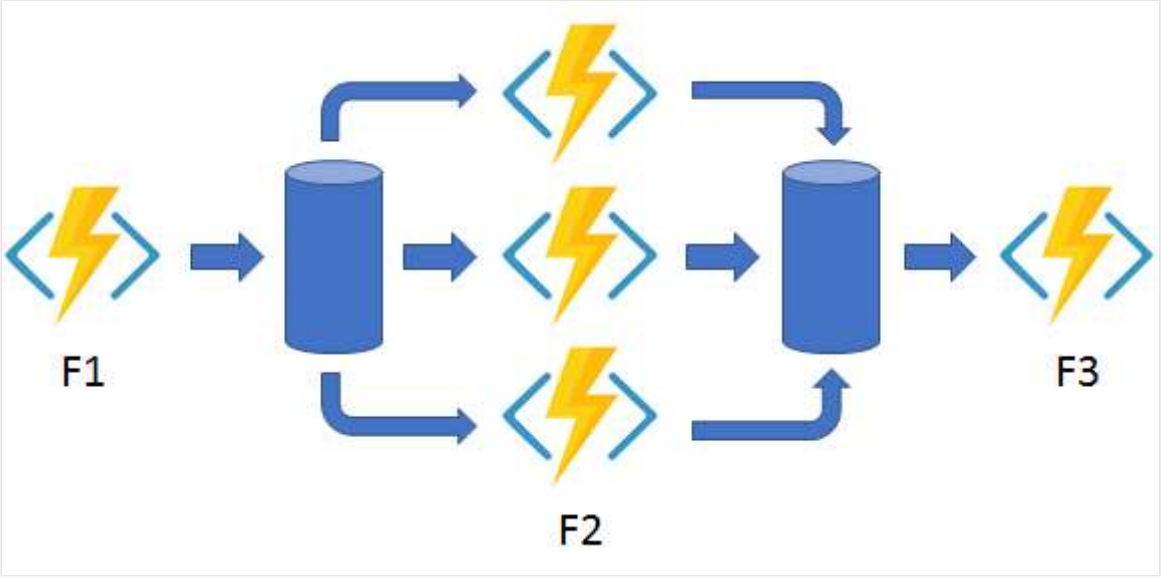
In the code example below, the values `F1`, `F2`, `F3`, and `F4` are the names of other functions in the function app. You can implement control flow by using normal imperative coding constructs. Code executes from the top down. The code can involve existing language control flow semantics, like conditionals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

C# Copy

```
[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.
    }
}
```

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. To fan in you write code to track when the queue-triggered functions end, and then store function outputs.

In the code example below, the fan-out work is distributed to multiple instances of the `F2` function. The work is tracked by using a dynamic list of tasks. The .NET `Task.WhenAll` API or JavaScript `context.df.Task.all` API 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.

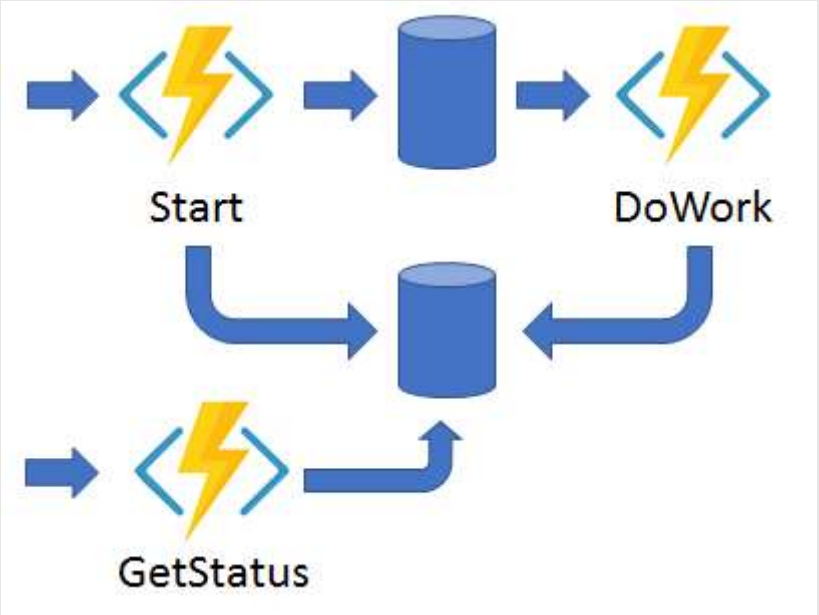
C# Copy

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

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. 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.

Copy

```
> curl -X POST https://myfunc.azurewebsites.net/orchestrators/DoWork -H "Content-
Length: 0" -i
HTTP/1.1 202 Accepted
Content-Type: application/json
Location:
https://myfunc.azurewebsites.net/runtime/webhooks/durabletask/b79baf67f717453ca9e86c5
da21e03ec
```

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

> curl
https://myfunc.azurewebsites.net/runtime/webhooks/durabletask/b79baf67f717453ca9e86c5
da21e03ec -i
HTTP/1.1 202 Accepted
Content-Type: application/json
Location:
https://myfunc.azurewebsites.net/runtime/webhooks/durabletask/b79baf67f717453ca9e86c5
da21e03ec

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

> curl
https://myfunc.azurewebsites.net/runtime/webhooks/durabletask/b79baf67f717453ca9e86c5
da21e03ec -i
HTTP/1.1 200 OK
Content-Length: 175
Content-Type: application/json

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

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.

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

C# Copy

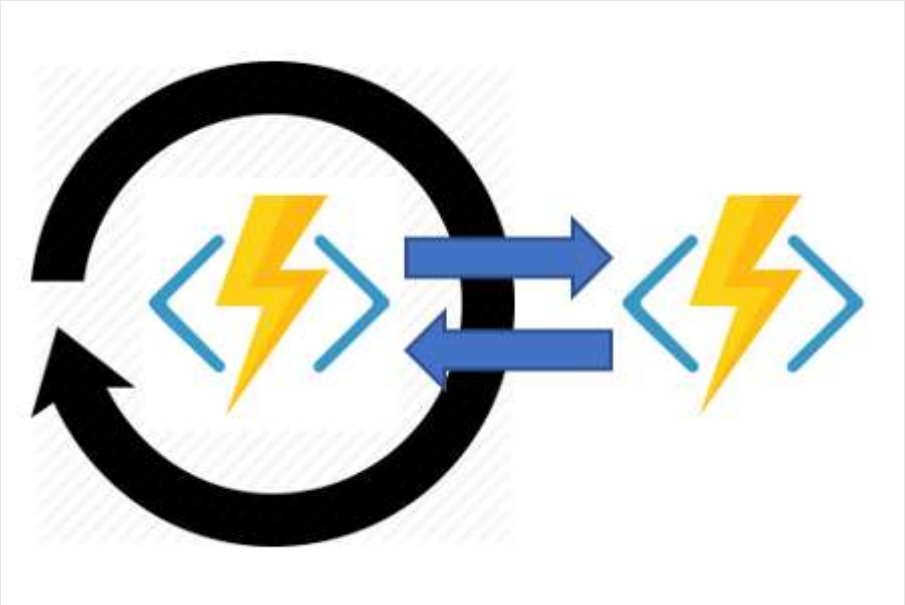
```
public static class HttpStart
{
    [FunctionName("HttpStart")]
    public static async Task<HttpResponseMessage> Run(
        [HttpTrigger(AuthorizationLevel.Function, methods: "post", Route =
"orchestrators/{functionName}")] HttpRequestMessage req,
        [DurableClient] IDurableClient starter,
        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.

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.



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# Copy

```
[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", jo-
bId);
        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.
}
```

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.

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.

C# Copy

```
[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);
        }
    }
}
```

Additional resources

- To learn about the differences between Durable Functions 1.x and 2.x visit:
  - Durable Functions versions overview

Next unit: Discover the four function types

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Discover the four function types

3 minutes

There are currently four durable function types in Azure Functions: orchestrator, activity, 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 the previous unit. 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.

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.

> **Note**
>
> The **Orchestrator function code constraints** article has detailed information on this requirement.

Activity functions

Activity functions are the basic unit of work in a durable function orchestration. 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.

An activity trigger is used to define an activity function. .NET functions receive a `DurableActivityContext` as a parameter. You can also bind the trigger to any other JSON-serializeable 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.

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. Some things to note:

- 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.
- Operations on entities require that you specify the **Entity ID** of the target entity, and the **Operation name**, which is a string that specifies the operation to perform.

Client functions

Orchestrator and entity functions are triggered by their bindings and both of these triggers work by reacting to messages that are enqueued in 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*.

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.

Next unit: Explore task hubs

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Explore task hubs

3 minutes

A task hub in Durable Functions is a logical container for durable storage resources that are used for orchestrations and entities. Orchestrator, activity, and entity functions can only directly interact with each other when they belong to the same task hub.

If multiple function apps share a storage account, each function app must be configured with a separate task hub name. A storage account can contain multiple task hubs. This restriction generally applies to other storage providers as well.

Azure Storage resources

A task hub in Azure Storage consists of the following resources:

- One or more control queues.
- One work-item queue.
- One history table.
- One instances table.
- One storage container containing one or more lease blobs.
- A storage container containing large message payloads, if applicable.

All of these resources are created automatically in the configured Azure Storage account when orchestrator, entity, or activity functions run or are scheduled to run.

Task hub names

Task hubs in Azure Storage are identified by a name that conforms 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:

JSON Copy

```
{
  "version": "2.0",
  "extensions": {
    "durableTask": {
      "hubName": "MyTaskHub"
    }
  }
}
```

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 unit: Explore durable orchestrations

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Explore durable orchestrations

3 minutes

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.

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.

> **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.

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.

Durable Functions uses event sourcing transparently. Behind the scenes, the `await` (C#) or `yield` (JavaScript) operator in an orchestrator function yields control of the orchestrator thread back to the Durable Task Framework dispatcher. 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 appends to the execution history of the orchestration instance. The history is stored in a storage table. The commit action then adds messages to a queue to schedule the actual work. At this point, the orchestrator function can be unloaded from memory.

When an orchestration function is given more work to do, 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.

Features and patterns

The table below describes the features and patterns of orchestrator functions.

Pattern/Feature	Description
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.

Pattern/Feature	Description
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 `Thread.Sleep` and `Task.Delay` (C#) or `setTimeout()` and `setInterval()` (JavaScript).
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.
Error handling	Orchestrator functions can use the error-handling features of the programming language. Existing patterns like `try/catch` are supported in orchestration code.
Critical sections	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.
Calling HTTP endpoints	Orchestrator functions aren't permitted to do I/O. 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.
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 to use ValueTuples objects in .NET.

Next unit: Control timing in Durable Functions

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Control timing in Durable Functions

3 minutes

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 `Thread.Sleep` and `Task.Delay` (C#), or `setTimeout()` and `setInterval()` (JavaScript), or `time.sleep()` (Python).

You create a durable timer by calling the `CreateTimer` (.NET) method or the `createTimer` (JavaScript) method of the orchestration trigger binding. The method returns a task that completes on a specified date and time.

Timer limitations

When you create a timer that expires at 4:30 pm, the underlying Durable Task Framework enqueues a message that becomes visible only at 4:30 pm. When running in the Azure Functions Consumption plan, the newly visible timer message will ensure that the function app gets activated on an appropriate VM.

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# Copy

```
[FunctionName("BillingIssuer")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    for (int i = 0; i < 10; i++)
    {
        DateTime deadline = context.CurrentUtcDateTime.Add(TimeSpan.FromDays(i + 1));
        await context.CreateTimer(deadline, CancellationToken.None);
        await context.CallActivityAsync("SendBillingEvent");
```

```
    }
}
```

Usage for timeout

This example illustrates how to use durable timers to implement timeouts.

C# Copy

```
[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;
        }
    }
}
```

> ⚠ **Warning**
>
> Use a `CancellationTokenSource` to cancel a durable timer (.NET) or call `cancel()` on the returned `TimerTask` (JavaScript) if your code will not wait for it to complete. The Durable Task Framework will not change an orchestration's status to "completed" until all outstanding tasks are completed or canceled.

This cancellation mechanism doesn't terminate in-progress activity function or sub-orchestration executions. It simply allows the orchestrator function to ignore the result and move on.

Next unit: Send and wait for events

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Send and wait for events

3 minutes

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.

Wait for events

The `WaitForExternalEvent` (.NET), `waitForExternalEvent` (JavaScript), and `wait_for_external_event` (Python) methods of the orchestration trigger binding allows an orchestrator function to asynchronously wait and listen for an external event. The listening orchestrator function declares the *name* of the event and the *shape of the data* it expects to receive.

The following example listens for a specific single event and takes action when it's received.

C# Copy

```
[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
    }
}
```

Send events

The `RaiseEventAsync` (.NET) or `raiseEvent` (JavaScript) method of the orchestration client binding sends the events that `WaitForExternalEvent` (.NET) or `waitForExternalEvent` (JavaScript) waits for. The `RaiseEventAsync` method takes *eventName* and *eventData* as parameters. The event data must be JSON-serializable.

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# Copy

```
[FunctionName("ApprovalQueueProcessor")]
public static async Task Run(
    [QueueTrigger("approval-queue")] string instanceId,
    [DurableClient] IDurableOrchestrationClient client)
{
    await client.RaiseEventAsync(instanceId, "Approval", true);
}
```

Internally, `RaiseEventAsync` (.NET) or `raiseEvent` (JavaScript) 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 queue. If the orchestration instance later begins listening for that *event name,* it will check the queue for event messages.

Next unit: Knowledge check

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

200 XP

Knowledge check

3 minutes

Check your knowledge

1. Which of the following durable function types is used to read and update small pieces of state?

- ○ Orchestrator
- ○ Activity
- ◉ Entity

✔ That's correct. Entity functions define operations for reading and updating small pieces of state.

2. Which application pattern would you use for a durable function that is polling a resource until a specific condition is met?

- ○ Function chaining
- ◉ Fan out/fan in

✘ That's incorrect. In the fan out/fan in pattern, you execute multiple functions in parallel and then wait for all functions to finish.

- ○ Monitor

✔ That's correct. The monitor pattern refers to a flexible, recurring process in a workflow. An example is polling until specific conditions are met.

Next unit: Summary

Continue

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Summary

3 minutes

In this module, you learned how to:

- Describe the app patterns typically used in Durable Functions
- Describe the four durable function types
- Explain the function Task Hubs perform in Durable Functions
- Describe the use of durable orchestrations, timers, and events

Module complete:

Review your Learning Path history

Explore other paths

How are we doing? ☆ ☆ ☆ ☆ ☆