**Durable Functions**

Durable Functions is a feature in Microsoft's Azure cloud platform that makes it easier to build complex tasks or workflows using serverless functions. It helps you create a sequence of steps, where each step depends on the result of the previous one. Durable Functions take care of managing the state, progress, and restarts for you, so you can focus on writing the business logic for each step, Allows you to build reliable and scalable applications without worrying about server management.

Durable Functions is an extension of [Azure Functions](https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview) that lets you write stateful functions in a serverless compute environment. The extension lets you define stateful workflows by writing [orchestrator functions](https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-orchestrations) and stateful entities by writing [entity functions](https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities) 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.

**1. Entity Functions:**

Entity Functions are a unique type of function in Azure Durable Functions that enable developers to create and manage durable, stateful entities. Unlike regular functions, which are stateless and run independently, Entity Functions allow you to maintain state across multiple function invocations. This stateful behavior is crucial for scenarios where you need to manage long-running operations or entities that retain their data over time.

In practical terms, an Entity Function represents a durable instance of an object that can be interacted with through function invocations. This object's state persists across function calls, allowing you to read, write, and update the entity's data.

**Use cases:**

Entity Functions are ideal for various use cases where you need to maintain stateful data for long-lived entities or processes. Some examples include:

a. Shopping Cart: In an e-commerce application, you can use an Entity Function to represent a user's shopping cart. The stateful nature of the Entity Function allows you to add, remove, and update items in the cart across multiple interactions, ensuring the cart's content persists over time.

b. Game Progress: In a gaming application, you can use Entity Functions to represent game sessions or player profiles. This way, you can save and update the player's progress, scores, and achievements as they play the game.

c. Workflow Approvals: For approval processes in business workflows, an Entity Function can represent an approval request, allowing multiple users to interact with the request and track its status over time.

**2. Orchestrator Functions:**

Orchestrator Functions are a critical component of Azure Durable Functions that serve as the central coordination point for complex workflows. They define the flow of execution for multiple functions, making it easier to create sophisticated and multi-step workflows.

An Orchestrator Function is designed to coordinate the execution of multiple functions in a specific order, managing the dependencies and ensuring that each function is executed at the right time. This is particularly useful in scenarios where you need to handle complex business processes or workflows that involve multiple steps and have dependencies between functions.

The workflow defined by an Orchestrator Function is executed in an orchestrated manner, allowing you to model and manage the sequence of actions precisely. It can handle decisions, branching, retries, and error handling, making it a powerful tool for building resilient and reliable workflows.

**Use cases:**

Orchestrator Functions are well-suited for various use cases that involve coordinating multiple functions and managing complex workflows. Some examples include:

a. Order Processing: In an e-commerce application, an Orchestrator Function can coordinate the sequence of functions to handle order processing, including inventory management, payment processing, and sending order confirmation emails.

b. Data Processing Pipelines: For data processing tasks, an Orchestrator Function can manage a series of functions to ingest, transform, and analyze data in a specific sequence, ensuring a reliable and efficient data pipeline.

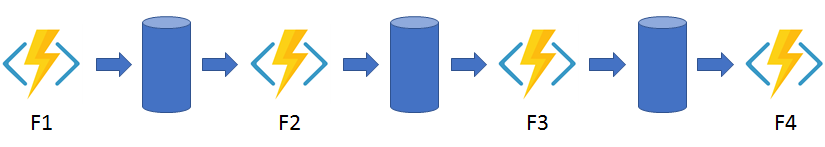
c. Business Workflows: Orchestrator Functions are useful for modeling and executing complex business workflows that involve human interactions, approvals, and multiple decision points.

In summary, Entity Functions and Orchestrator Functions in Azure Durable Functions provide powerful capabilities to manage stateful entities and orchestrate complex workflows. Entity Functions enable the persistence of data across function invocations, while Orchestrator Functions act as the central coordinator for multiple functions, allowing you to create sophisticated, reliable, and scalable workflows. Together, these features empower developers to build long-running, stateful applications with ease and flexibility.

**Application Pattern:**

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](https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp-inproc" \l "chaining)
* [Fan-out/fan-in](https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp-inproc" \l "fan-in-out)
* [Async HTTP APIs](https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp-inproc" \l "async-http)
* [Monitoring](https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp-inproc" \l "monitoring)
* [Human interaction](https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp-inproc" \l "human)
* [Aggregator (stateful entities)](https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp-inproc" \l "aggregator)
* **Pattern #1: Function chaining**



Function chaining is a way to connect multiple functions together in a specific order, where the output of one function becomes the input for the next function. Imagine you have a series of tasks, and each task requires the result of the previous task to work correctly. Function chaining allows you to ensure that these tasks execute in the right sequence.

For example, let's consider a scenario where you want to process some data through a sequence of functions: F1, F2, F3, and F4. Each function performs a specific task on the data and passes the result to the next function in line. Here's how it works:

1. Function F1 is the starting point. It takes some initial data as input and processes it. The output of F1 becomes the input for F2.

2. Function F2 receives the output from F1, performs its task on the data, and produces a new result. This result is then passed to F3.

3. Function F3 takes the output of F2, processes it, and generates a new output that will be sent to F4.

4. Finally, Function F4 receives the output from F3, performs the last processing step, and produces the final result.

Now, Durable Functions come into play to implement this function chaining pattern seamlessly. Durable Functions allow you to define an orchestrator function, which acts as the conductor of this entire sequence. The orchestrator function calls each individual function (F1, F2, F3, and F4) in the required order and passes the results between them.

By using queues internally, Durable Functions ensure that the sequence remains durable and scalable, meaning the system can handle large amounts of data and maintain the order of execution even if there are failures or delays.

The orchestrator function can also include regular programming constructs like conditions (if statements) and loops to control the flow of execution based on the intermediate results. Additionally, you can handle errors using try/catch/finally blocks to handle exceptions gracefully.

The function chaining pattern using Durable Functions allows you to execute a series of functions in a specific order, passing the output of one function to the next. This approach provides durability, scalability, and control flow capabilities, enabling you to process data efficiently and reliably through a sequence of tasks.

**Java Code:**

**@FunctionName("Chaining")**

**public double functionChaining(**

**@DurableOrchestrationTrigger(name = "ctx") TaskOrchestrationContext ctx) {**

**String input = ctx.getInput(String.class);**

**int x = ctx.callActivity("F1", input, int.class).await();**

**int y = ctx.callActivity("F2", x, int.class).await();**

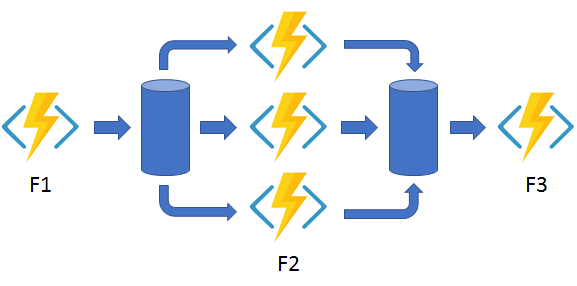
**int z = ctx.callActivity("F3", y, int.class).await();**

**return ctx.callActivity("F4", z, double.class).await();**

**}**

You can use the ctx object to invoke other functions by name, pass parameters, and return function output. The output of these method calls is a Task<V> object where V is the type of data returned by the invoked function. Each time you call Task<V>.await(), the Durable Functions framework checkpoints the progress of the current function instance. If the process unexpectedly recycles midway through the execution, the function instance resumes from the preceding Task<V>.await() call

* **Pattern #2: Fan out/fan in**



Imagine you have a list of tasks that you want to process, but each task takes some time to complete. Instead of processing these tasks one by one, which would be slow, you want to process them in parallel to speed things up. This is the "fan-out" part of the pattern.

To do this, you can create a function (let's call it the "fan-out function") that takes the list of tasks as input. Inside this function, you loop through the tasks and send each task as a separate message to a queue. So, you're "fanning out" the tasks to the queue, allowing them to be processed in parallel.

Now, the challenge is that you need to wait for all the tasks to complete before proceeding further, as you may need to aggregate or combine the results of these tasks. This is the "fan-in" part of the pattern.

To "fan back in," you create another function (let's call it the "fan-in function") that is triggered by the completion of each task in the queue. When a task completes, the queue triggers the "fan-in function" and provides the output of that task as a message to the function.

Inside the "fan-in function," you keep track of how many tasks have completed and store their outputs in a suitable data structure, such as an array or a dictionary. You can also check if all the tasks have completed by comparing the number of completed tasks with the total number of tasks you initially sent to the queue.

Once all tasks are completed, you can perform any necessary aggregation or processing on the collected outputs. After the aggregation work is done, the "fan-in function" can produce the final result or take any further action based on the combined results of all the tasks.

In summary, the fan-out/fan-in pattern allows you to process multiple tasks in parallel by sending them to a queue for processing and then collecting their results as they complete. This enables more efficient processing and better utilization of resources in scenarios where tasks can be done independently and concurrently.

**Java Code:**

**@FunctionName("FanOutFanIn")**

**public Integer fanOutFanInOrchestrator(**

**@DurableOrchestrationTrigger(name = "ctx") TaskOrchestrationContext ctx) {**

**// Get the list of work-items to process in parallel**

**List<?> batch = ctx.callActivity("F1", List.class).await();**

**// Schedule each task to run in parallel**

**List<Task<Integer>> parallelTasks = batch.stream()**

**.map(item -> ctx.callActivity("F2", item, Integer.class))**

**.collect(Collectors.toList());**

**// Wait for all tasks to complete, then return the aggregated sum of the results**

**List<Integer> results = ctx.allOf(parallelTasks).await();**

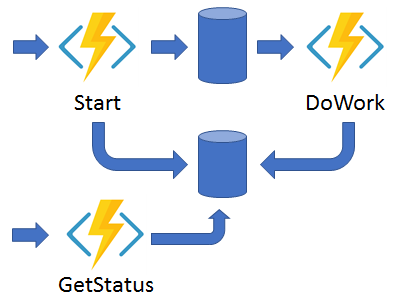
**return results.stream().reduce(0, Integer::sum);**

**}**

The fan-out work is distributed to multiple instances of the F2 function. The work is tracked by using a dynamic list of tasks. ctx.allOf(parallelTasks).await() is called to wait for all the called functions to finish. Then, the F2 function outputs are aggregated from the dynamic task list and returned as the orchestrator function's output.

The automatic checkpointing that happens at the .await() call on ctx.allOf(parallelTasks) ensures that an unexpected process recycle doesn't require restarting any already completed tasks.

* **Pattern #3: Async HTTP APIs**



Imagine you have a long-running operation or task that you want to perform. Instead of making the client wait for the operation to complete, you want to trigger the operation and provide a way for the client to check its status later. This is where the async HTTP API pattern comes in.

1. \*\*Triggering the Long-Running Operation:\*\* To start the long-running operation, you create an HTTP endpoint that the client can call. When the client makes a request to this endpoint, the operation is initiated, and the server responds immediately with a confirmation, allowing the client to proceed without waiting for the operation to finish.

2. \*\*Redirecting to a Status Endpoint:\*\* Along with the confirmation, the server provides the client with a URL to a status endpoint. The client can use this URL to check the status of the operation later.

3. \*\*Polling the Status Endpoint:\*\* After the client receives the status endpoint URL, it can periodically make requests to this endpoint (polling) to inquire about the progress or completion of the long-running operation. The server responds with the current status of the operation, allowing the client to determine if it's still in progress or if it has finished.

In Durable Functions, this pattern is made even easier to implement. When you trigger the long-running operation using Durable Functions, it automatically generates the status endpoint URLs for you. These status endpoints are exposed as webhook HTTP APIs. So, after you start the operation using the Durable Functions API, the client can simply use the provided status endpoint URLs to check the status without you having to write additional code to manage this process.

Overall, the async HTTP API pattern with Durable Functions allows you to start long-running operations efficiently and provides a simple way for clients to check their status without waiting, making it ideal for coordinating stateful and time-consuming tasks.

Durable Functions automates the state management for long-running tasks, so you don't have to create your own status-tracking mechanism.

Durable Functions provides built-in HTTP APIs that handle the management of long-running tasks for you. Alternatively, you can implement this pattern yourself by using different function triggers like HTTP, queues, or Azure Event Hubs, along with the durable client binding.

For instance, you could use a queue message to trigger the termination of a long-running task. Alternatively, you might create an HTTP trigger protected by Azure Active Directory authentication instead of using the built-in HTTP APIs that rely on a generated key for authentication.

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#](https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-create-first-csharp), [JavaScript](https://learn.microsoft.com/en-us/azure/azure-functions/durable/quickstart-js-vscode), [TypeScript](https://learn.microsoft.com/en-us/azure/azure-functions/durable/quickstart-ts-vscode), [Python](https://learn.microsoft.com/en-us/azure/azure-functions/durable/quickstart-python-vscode), [PowerShell](https://learn.microsoft.com/en-us/azure/azure-functions/durable/quickstart-powershell-vscode), and [Java](https://learn.microsoft.com/en-us/azure/azure-functions/durable/quickstart-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**

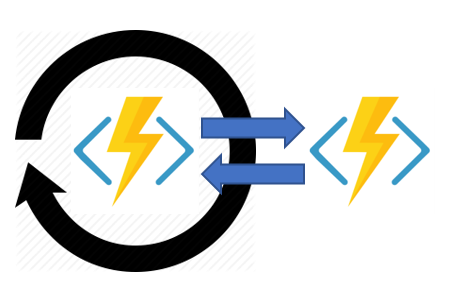
**Content-Type: application/json**

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

* **Pattern #4: Monitor**

The monitor pattern is a method used in workflows to handle flexible and repetitive tasks. It involves checking for specific conditions until they are met. While a basic scenario can be addressed using a regular timer trigger, it has limitations with fixed intervals and complex management of instances. Durable Functions, however, offer a solution by allowing the creation of flexible recurrence intervals, managing task lifetimes, and enabling the creation of multiple monitor processes within a single orchestration.

For example, let's consider an asynchronous HTTP API scenario. Instead of having an external client monitor a long-running operation through an endpoint, the monitor pattern reverses this approach. The long-running monitor now consumes an external endpoint and waits for a state change to occur. This way, the monitor can keep checking the external endpoint until the desired state is reached, providing a more efficient and flexible way to manage long-running operations.



**Java Code:**

**@FunctionName("Monitor")**

**public String monitorOrchestrator(**

**@DurableOrchestrationTrigger(name = "ctx") TaskOrchestrationContext ctx) {**

**JobInfo jobInfo = ctx.getInput(JobInfo.class);**

**String jobId = jobInfo.getJobId();**

**Instant expiryTime = jobInfo.getExpirationTime();**

**while (ctx.getCurrentInstant().compareTo(expiryTime) < 0) {**

**String status = ctx.callActivity("GetJobStatus", jobId, String.class).await();**

**// Perform an action when a condition is met**

**if (status.equals("Completed")) {**

**// send an alert and exit**

**ctx.callActivity("SendAlert", jobId).await();**

**break;**

**}**

**// wait N minutes before doing the next poll**

**Duration pollingDelay = jobInfo.getPollingDelay();**

**ctx.createTimer(pollingDelay).await();**

**}**

**return "done";**

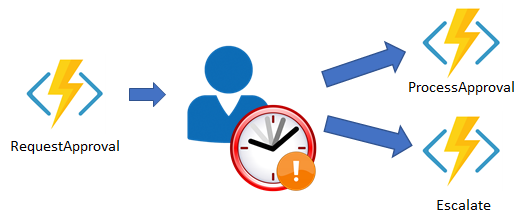
**}**

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](https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-timers) to request approval. The orchestrator escalates if timeout occurs. The orchestrator waits for an [external event](https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-external-events), such as a notification that's generated by a human interaction.

**Java Code:**

**@FunctionName("ApprovalWorkflow")**

**public void approvalWorkflow(**

**@DurableOrchestrationTrigger(name = "ctx") TaskOrchestrationContext ctx) {**

**ApprovalInfo approvalInfo = ctx.getInput(ApprovalInfo.class);**

**ctx.callActivity("RequestApproval", approvalInfo).await();**

**Duration timeout = Duration.ofHours(72);**

**try {**

**// Wait for an approval. A TaskCanceledException will be thrown if the timeout expires.**

**boolean approved = ctx.waitForExternalEvent("ApprovalEvent", timeout, boolean.class).await();**

**approvalInfo.setApproved(approved);**

**ctx.callActivity("ProcessApproval", approvalInfo).await();**

**} catch (TaskCanceledException timeoutEx) {**

**ctx.callActivity("Escalate", approvalInfo).await();**

**}**

**}**

The ctx.waitForExternalEvent(...).await() method call pauses the orchestration until it receives an event named ApprovalEvent, which has a boolean payload. If the event is received, an activity function is called to process the approval result. However, if no such event is received before the timeout (72 hours) expires, a TaskCanceledException is raised and the Escalate activity function is called.

An external client can deliver the event notification to a waiting orchestrator function by using the [built-in HTTP APIs](https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-http-api" \l "raise-event):

**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:

**Java Code:**

**@FunctionName("RaiseEventToOrchestration")**

**public void raiseEventToOrchestration(**

**@HttpTrigger(name = "instanceId") String instanceId,**

**@DurableClientInput(name = "durableContext") DurableClientContext durableContext) {**

**DurableTaskClient client = durableContext.getClient();**

**client.raiseEvent(instanceId, "ApprovalEvent", true);**

**}**

* **Pattern #6: Aggregator (stateful entities):**

The sixth pattern involves aggregating event data over a period of time into a single, organized entity. This data may come from various sources, arrive in batches, or be collected over extended periods. The aggregator may need to take actions on the data as it comes in, and external clients might want to query the aggregated data.

Implementing this pattern with normal, stateless functions can be tricky due to concurrency control challenges. Dealing with multiple threads modifying the same data simultaneously and ensuring the aggregator runs on only one virtual machine (VM) at a time becomes a significant issue.

In simpler terms, this pattern is about collecting and combining data from different sources or events over time into a single place for easier access and analysis. The challenge arises when we try to do this using regular functions, as handling multiple threads and ensuring only one aggregator runs at a time becomes complex. However, using Durable Functions, we can overcome these challenges and efficiently implement this pattern.

* **Technology:**

Durable Functions is an extension of Azure Functions, designed to help build complex workflows and manage long-running operations in a serverless environment. It is built on top of the Durable Task Framework, an open-source library on GitHub. Just as Azure Functions is the serverless evolution of Azure WebJobs, Durable Functions is the serverless evolution of the Durable Task Framework.

The Durable Task Framework is widely used by Microsoft and other organizations to automate critical processes. It provides a way to create workflows in code, making it easier to manage and orchestrate tasks over time. This powerful framework is a natural fit for the serverless environment of Azure Functions, enabling developers to create efficient and scalable workflows without worrying about infrastructure management.

* **Durable Task Framework:**

The Durable Task Framework is an open-source library developed by Microsoft. It is designed to simplify the development of complex, long-running, and stateful workflows in distributed systems. The framework provides a set of tools and APIs to build and manage asynchronous, durable operations, making it easier to create workflows that span multiple function invocations and maintain state across them.

The key features of the Durable Task Framework include:

1. Orchestration: It allows developers to define workflows as code, specifying the sequence of steps and their dependencies. Workflows can involve multiple tasks and activities, and the framework ensures that they are executed in the correct order.

2. State Management: The framework handles the storage and management of workflow state, ensuring that the state is durable and can be persisted across function invocations. This is essential for long-running operations that need to keep track of their progress.

3. Timers and Delays: The Durable Task Framework supports timers and delays, enabling developers to schedule activities or wait for specific periods before executing the next steps in the workflow.

4. Human Interaction: It also supports human interaction patterns, allowing workflows to pause and wait for input from users or external systems before continuing.

5. Retry and Error Handling: The framework includes built-in capabilities for retrying failed tasks and handling errors to ensure reliable execution of workflows.

6. Scalability: The Durable Task Framework is designed to be scalable, making it suitable for handling workflows with high throughput and large numbers of concurrent instances.

Overall, the Durable Task Framework simplifies the development of complex and long-running workflows in distributed systems, making it easier for developers to build robust and scalable applications that require coordination and state management across multiple function invocations.