# Azure Durable Functions

SUCCINCTLY

*BY* **MASSIMO BONANNI**

SUCCINCTLY EBOOK SERIES

Syncfusion®

# Azure Durable Functions Succinctly

**By**

**Massimo Bonanni**

Foreword by Daniel Jebaraj

ISBN: 978-1-64200-221-8

**Technical Reviewer:** James McCaffrey
**Copy Editor:** Courtney Wright
**Acquisitions Coordinator:** Tres Watkins, VP of content, Syncfusion, Inc.
**Proofreader:** Jacqueline Bieringer, content producer, Syncfusion, Inc.

# Table of Contents

# The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, CEO
Syncfusion, Inc.

## Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

## Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

## The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

## The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

## Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

## Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

## Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

# About the Author

I'm an Azure technical trainer at Microsoft, and my mission is to empower customers and partners to achieve more by leveraging the power of Azure in their solutions.

I specialize in cloud application development and DevOps methodologies and tools, and I worked for several years in desktop client development with Visual Studio in the past. I like to share my knowledge with other people because sharing knowledge is a great way to grow: you can teach what you know, but you can learn from others, too—and have fun while doing it.

I work in an EMEA team (Europe, Middle East, Asia) for a worldwide organization that provides thousands of courses each year in different languages, and for thousands of customers and partners. I'm a Microsoft Certified Trainer and a technical speaker in several conferences in Italy and Europe (and Microsoft internal events). I founded two user groups in Italy and managed them for several years.

I’m also passionate about biking, reading, and dogs!

## Let’s connect

Feel free to connect with me on LinkedIn or Twitter. Or send me an email and give me feedback on this book: what parts you like and what can I improve.

- Twitter: https://twitter.com/massimobonanni
- LinkedIn: https://www.linkedin.com/in/massimobonanni/
- Email: massimo.bonanni@tiscali.it

# Chapter 1 Azure Functions Recap

## Serverless architecture

Serverless architecture is one of the trendy architectures of the moment, and sometimes people use fashionable architecture just because it is fashionable, and not because it's needed.

Using the wrong architecture means having problems that you wouldn't have if you'd chosen the correct one. For this reason, the most important thing to understand before starting to use Azure Functions and Durable Functions is what serverless architecture is, and what kind of scenarios it addresses.

A serverless architecture (or the serverless technologies you can use to implement it) is based on the following pillars:

- Infrastructure abstraction
- Event-driven scalability
- Pay-as-you-go pricing model

Infrastructure abstraction means that you don't care about the infrastructure layer when your code runs. You write your code and deploy it on the platform, and the platform (Azure) chooses when your code runs. For this reason, serverless architecture is a PaaS (platform as a service) solution, and Azure gives you some technologies to implement it.

Event-driven scalability means that the platform scales your code (typically increasing or decreasing the number of the resources, behind the scenes, that host your code) as it needs, based on the number of events your solution receives. This consideration implies that you can use a serverless architecture when your solution manages events. If you don't have events in your scenario, serverless architecture is probably not the best choice.

The pay-as-you-go pricing model means that you pay only for the times your serverless code is executed. If you don't receive any events in your solution, you don't pay any money. On the other hand, if you receive one billion events and the platform scales to manage them, you pay for one billion of them.

To make an actual example, consider using Azure App Service. Azure App Service is a PaaS service provided by Microsoft Azure that allows you to host an HTTP application (like a website). App Service is not a serverless technology because:

- You don't manage the virtual machines (VM) behind the scenes of your app service instance, but you have to declare the power of those VMs. For this reason, you are not completely abstracted from the infrastructure that hosts your application behind the scenes.
- Even if your website manages HTTP calls, and you can consider an HTTP call as an event (someone wants to retrieve your home page), your App Service instance doesn't scale to manage any number of requests. The limit of the App Service is the App Service plan you use.
- You pay a monthly fee for the App Service plan you are using in your App Service, and you pay it even if nobody calls your website.

App Service doesn't match at least two of the three pillars of a serverless technology (event-driven scalability and pay-as-you-go pricing model). For this reason, App Service is not a serverless technology.

## Azure serverless technologies

Azure has several serverless technologies, and the most important are:

- Azure Functions
- Azure Logic Apps
- Event Grid

We will talk about Azure Functions shortly.

Azure Logic Apps is a PaaS service that allows you to create a workflow to integrate and orchestrate data between external systems. Azure Logic Apps gives you a graphic editor that allows you to build workflows using building blocks, called connectors, to process, retrieve, and send data to external services.

The following figure shows you a simple logic app that copies a file from DropBox in OneDrive. As you can see, the workflow has a construct "if" in its definition: the logic app copies all the files with a name ending in **.jpg** to a specific folder.

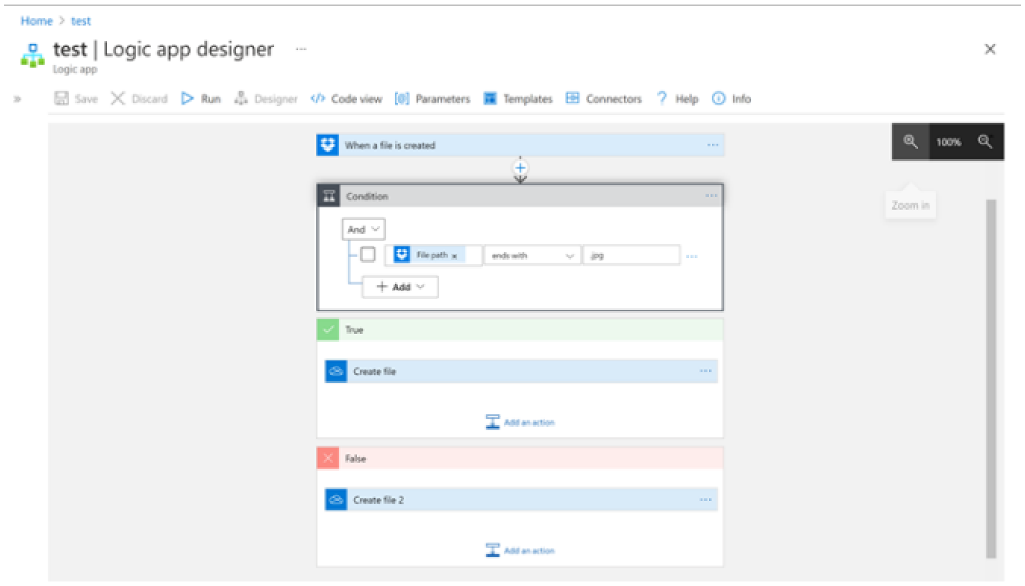


*Figure 1: A simple logic app that moves files between DropBox and OneDrive*

One of the most important components of logic apps is connectors. When writing this book, the Logic Apps platform has more than 400 connectors for several enterprise services like SAP, Oracle, and IBM 3270.

The following figure shows you some of the most important connectors provided by Logic Apps.

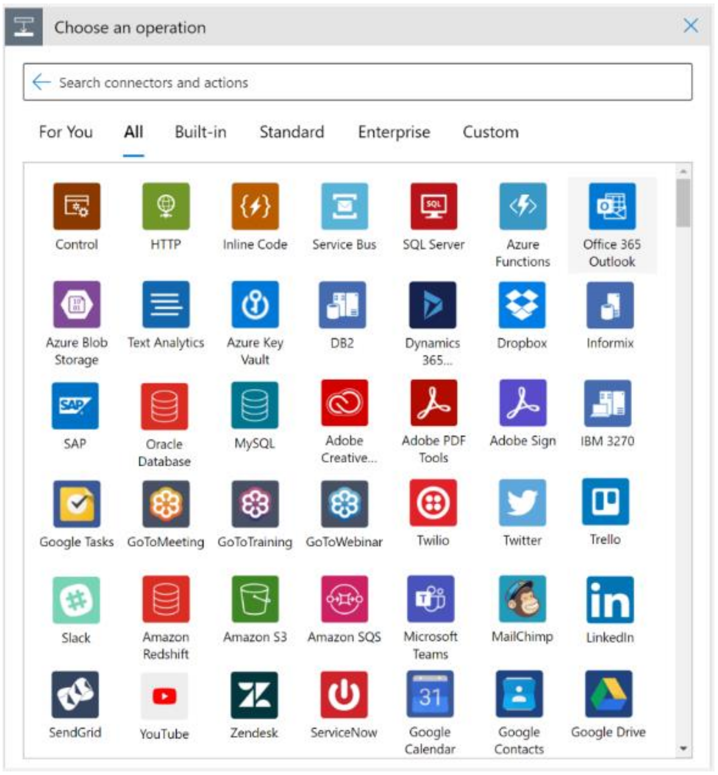


*Figure 2: Some of the most important connectors in Logic Apps*

Every connector allows you to interact with a specific external service. It can have one or more triggers and actions. A trigger is a component that starts your logic app when an event occurs, while an action allows you to interact with an external service, for example, to save or read data from a database or send an email using Office 365.

You can consider Logic Apps a low-code or no-code platform. You don't need to know a programming language to create your workflow. At this moment, the most important limitation is that a logic app can run only in Azure, but Microsoft is working to make Logic Apps portable, like Azure Functions.

Azure Event Grid is the Azure routing event engine. It allows you to publish an event from any source and route it through any other destination. Event Grid implements the publish/subscribe pattern.

When you want to route an event from a source to a destination, you create an event grid topic to connect the event producer to the routing engine, and an event grid subscription to propagate the event from the event routing engine to the destination (called the handler). You can have more than one subscription for each topic. Event Grid also gives you some filtering capabilities to select what types of events a particular handler needs to receive.

In the following image, you can see sources and handlers for Event Grid.

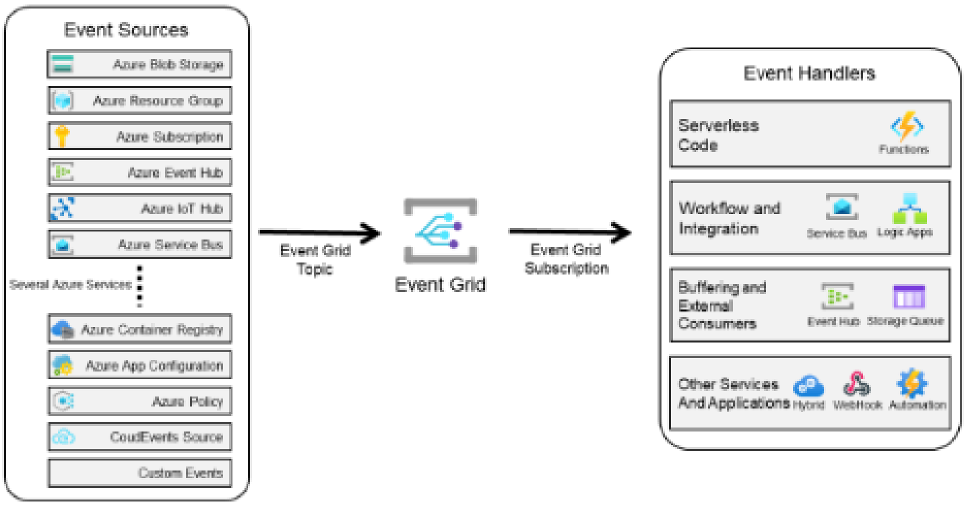


*Figure 3: Event Grid sources and handlers*

## Azure serverless scenarios

In this section, I would like to provide you a few interesting scenarios in which you can leverage the power of a serverless architecture.

### Web application order management

Here is a classic e-commerce scenario: when a customer buys something, the web application creates a message that contains the customer order and enqueues it in Service Bus. A function processes the order and writes the order data in a database.

The creation of the order is the event that triggers the whole process.

The schema of the scenario is represented in the following figure.

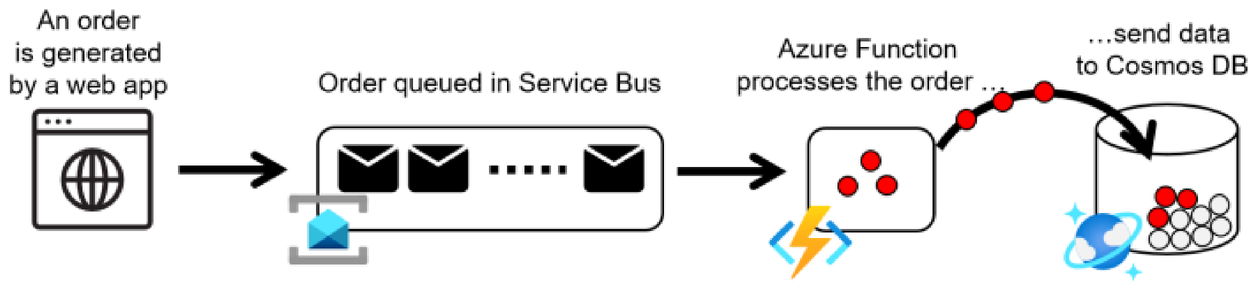

An order is generated by a web app
Order queued in Service Bus
Azure Function processes the order …
…send data to Cosmos DB

*Figure 4: The order management scenario*

## Mobile application backend

In this scenario, a mobile application calls a REST API exposed by one Azure function that validates and processes the request and writes the data in a database. (In the following figure, you can see Cosmos DB, but you can figure out your favorite database in the process). When the data is persisted in the database, another Azure function is triggered and sends a mobile notification to several devices using Notification Hub.

The complete schema is shown in the following figure.

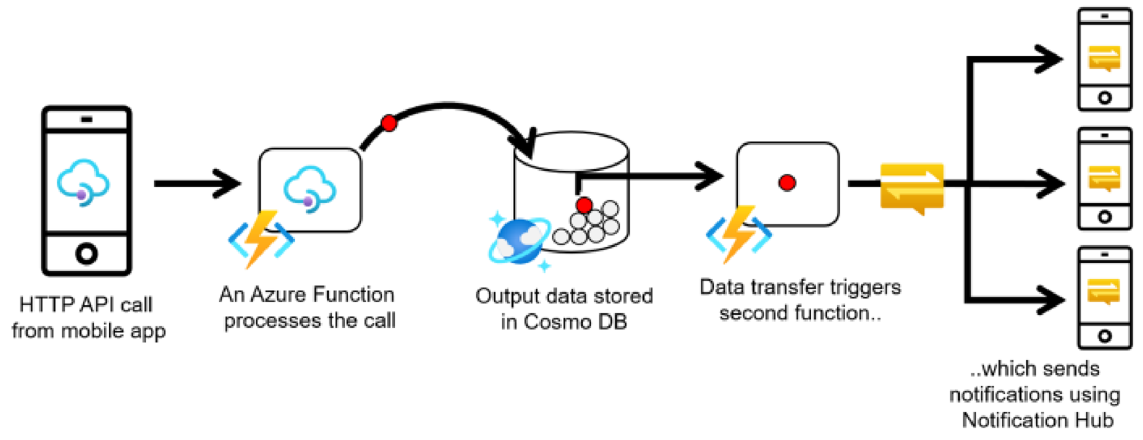


*Figure 5: Mobile application backend scenario*

## Real-time file processing

Suppose you have to analyze and process pictures of a street taken by a security camera to retrieve a car's plates in transit. The following image describes the scenario.

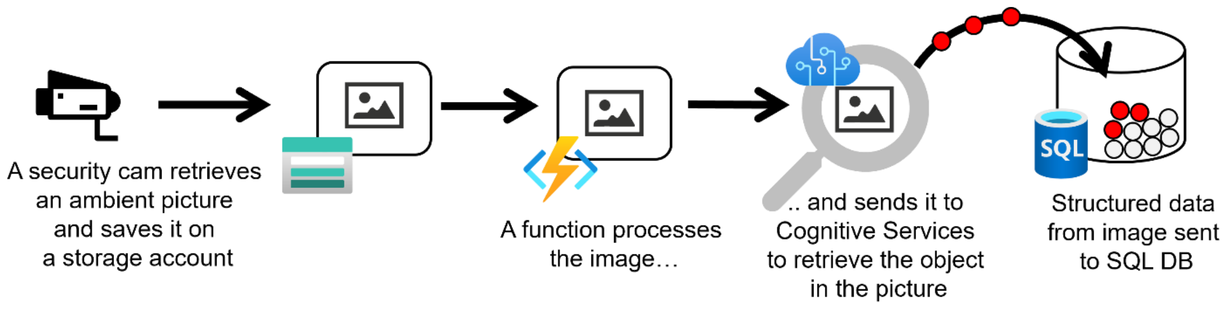


*Figure 6: Real-time file processing*

Every time the security camera takes a photo of the street (because, for example, a car passes in the field of view of the camera), it saves the photo in a storage account. An Azure function is triggered by the new image, calls Azure Cognitive Services to retrieve information about the car's plate in the picture, and then writes the data in a database.

### Scheduled task automation

Azure functions (and serverless technologies in general) are very helpful when you need to automate tasks. In this scenario, an Azure function runs every 15 minutes to clean data in a database (removing entries, deduplicating entries, and so on). The complete scenario is described in the following figure.

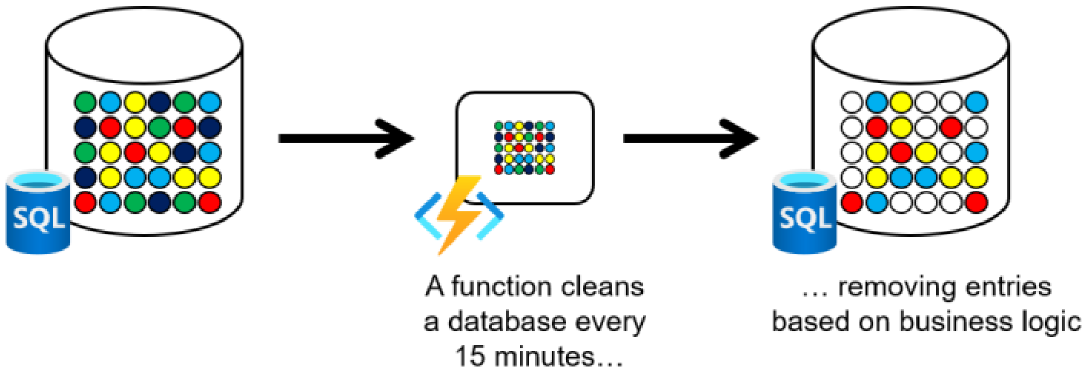

SQL
A function cleans a database every 15 minutes…
SQL
… removing entries based on business logic

*Figure 7: Scheduled task automation*

All the scenarios I've mentioned can be considered workflow. In every scenario, several functions need to be orchestrated to achieve the goal. Sometimes it is not easy to do using standard Azure functions to implement those scenarios, but fortunately, Durable Functions, as we will see in the following chapters, can simplify implementing them.

## What are Azure functions?

Azure Functions is one of the most important serverless technologies proposed by Azure. Even though this book is not directly about standard Azure functions, it is fundamental to recap what an Azure function is. As we will see in the following chapters, Durable Functions is based on Azure Functions, so all the basic behavior is the same.

An Azure function is literally a snippet of code that executes on a runtime called Azure Functions Runtime and can be written in several programming languages. In the following table, you'll find all the programming languages supported at this moment.

*Table 1: Programming languages supported by Azure Functions*

| Language | Supported Runtime |
|---|---|
| C# | .NET Core 3.1 (.NET Core 5.0 in preview) |
| JavaScript/TypeScript | Node 10 and 12 |
| F# | .NET Core 3.1 |
| Java | Java 8 |

| Language | Supported Runtime |
|---|---|
| Python | Python 3.6, 3.7, and 3.8 |
| PowerShell | PowerShell Core 6 |

Azure Functions Core Tools allows you to develop and test your functions on your local PC. It is a multiplatform (runs on Windows, Linux, and macOS) command-line tool, and you can download and install it following these instructions.

***Note: Azure Functions Core Tools also contains the Azure Functions Runtime, and that is the reason you can locally test your functions.***

Once you install the tool, it provides you with a set of commands to create, implement, test, and deploy Azure functions in one of the supported programming languages.

When you start implementing your Azure functions, you put all of them in a function app. Function app is the deployment unit for Azure Functions (that means you deploy all the functions in the same function app simultaneously in the same Azure service), and the scalability unit for Azure.

To create your function app, you create the folder in which you want to create the function app, move inside it, and use the following command.

```
func init --worker-runtime dotnet
```

The command generates all the files you need for a .NET project, but you have the same behavior if you want to use a different runtime (for example, Java).

After you have created the function app, you can start to create every single function inside it. To create a function, you can use another command provided by the tool.

```
func new --name HttpHelloWorld --language c# --template httptrigger
```

The previous command creates the file definition for a function called **`HttpHelloWorld`**, written in C# and with an **`HttpTrigger`** trigger.

Once you create your functions using the tool, you can use your preferred IDE to change your code and compile and debug your functions. In this book, I will use Visual Studio.

Before looking at the code generated by the tool, we must define some terminology used in the Azure functions. The following schema summarizes all the components involved in an Azure function.

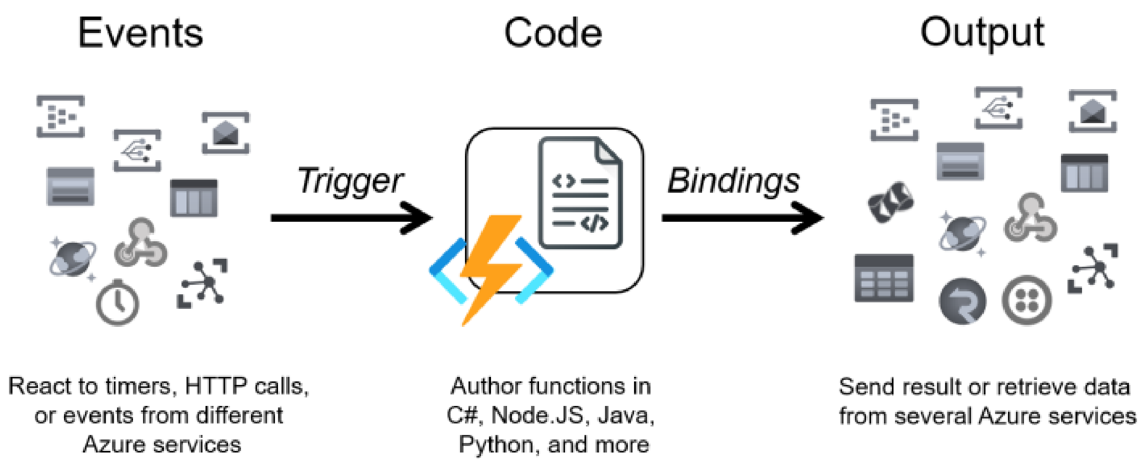

Events
Code
Output
Trigger
Bindings
React to timers, HTTP calls, or events from different Azure services
Author functions in C#, Node.JS, Java, Python, and more
Send result or retrieve data from several Azure services

*Figure 8: Azure functions components*

Because Azure Functions is a serverless technology, every function is triggered by an event, and the component in the runtime with the responsibility to do that is called the trigger. An Azure function must have one and only one trigger.

Inside your function code, you can interact with external services using other components provided by the runtime. They are called binding; each can be used to read data, send data, or both. A function can have zero, one, or multiple bindings.

Now, we can take a look at the class and method that define a simple function. The following snippet of code is the **`HttpHelloWorld`** generated by the commands executed earlier.

*Code Listing 1: Function code generated by the tool*

```
public static class HttpHelloWorld
{
    [FunctionName("HttpHelloWorld")]
    public static async Task<IActionResult> Run(
        [HttpTrigger(AuthorizationLevel.Function, "get", "post")]
HttpRequest req,
        ILogger log)
    {
        log.LogInformation("C# HTTP trigger function processed a
request.");

        string name = req.Query["name"];

        string requestBody = await new
StreamReader(req.Body).ReadToEndAsync();
        dynamic data = JsonConvert.DeserializeObject(requestBody);
        name = name ?? data?.name;

        string responseMessage = string.IsNullOrEmpty(name)
```

```
            ? "This HTTP triggered function executed successfully. Pass a 
name in the query string or in the request body for a personalized 
response."
            : $"Hello, {name}. This HTTP triggered function executed 
successfully.";

        return new OkObjectResult(responseMessage);
    }
}
```

The attribute **FunctionName**, before the method definition, declares that **Run** is your Azure function (and defines the name of the function). When the Azure Functions runtime starts and tries to host your functions, you have to remember that it searches for all the methods in your code with that attribute. No matter what the name of your function method, the runtime refers to it using the name declared in the attribute.

The **Run** method has two arguments. The type of the first one is an **HttpRequest**, and it is decorated by the attribute **HttpTrigger**. That attribute tells the runtime that your function has a particular trigger. In this case, the trigger is an **HttpTrigger**; that means your function will answer to an HTTP request.

As you can see, the attribute of the argument defines the type of trigger, while the type of the argument is the payload associated with the trigger itself.

The other method argument is the log instance you can use to trace information into the platform log (in the same way you do with a standard .NET application).

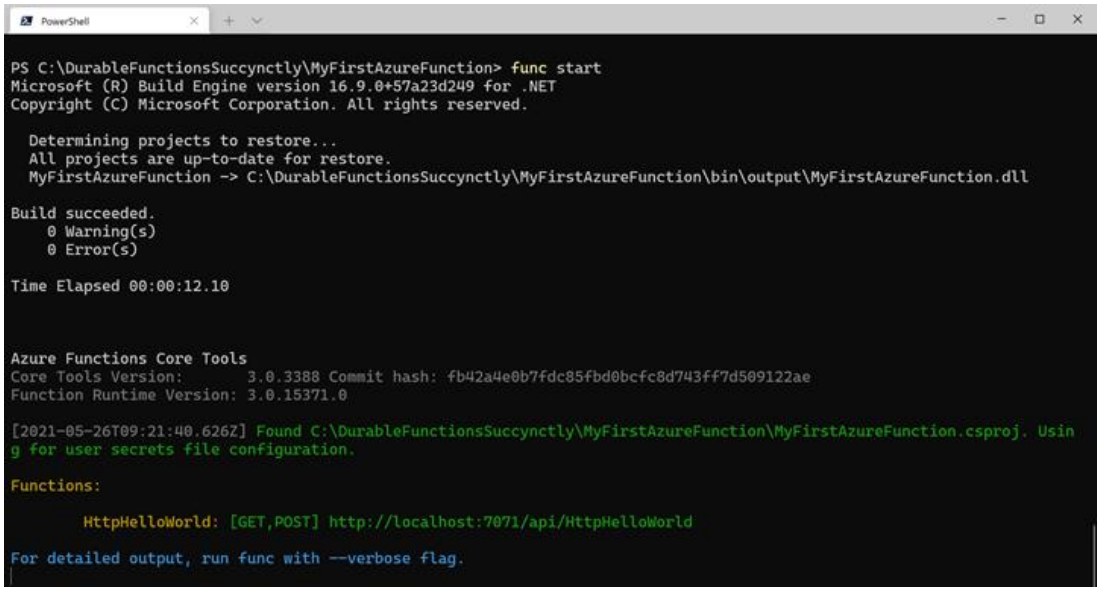
You don't have any binding declared in the method signature in the previous function, but the function returns an HTTP response. You can consider the return value as one of the possible types of binding.

You can test your function locally. Azure Functions Core Tools provides you the command to execute the runtime and host your compiled code. To do that, open a command prompt, move into the folder that contains your function app, and run the following command.

```
func start
```

This command executes the runtime and hosts your functions. When the runtime starts, it scaffolds for your code to find all your functions, tries to resolve all triggers and bindings you use in your code, and finally, runs your functions.

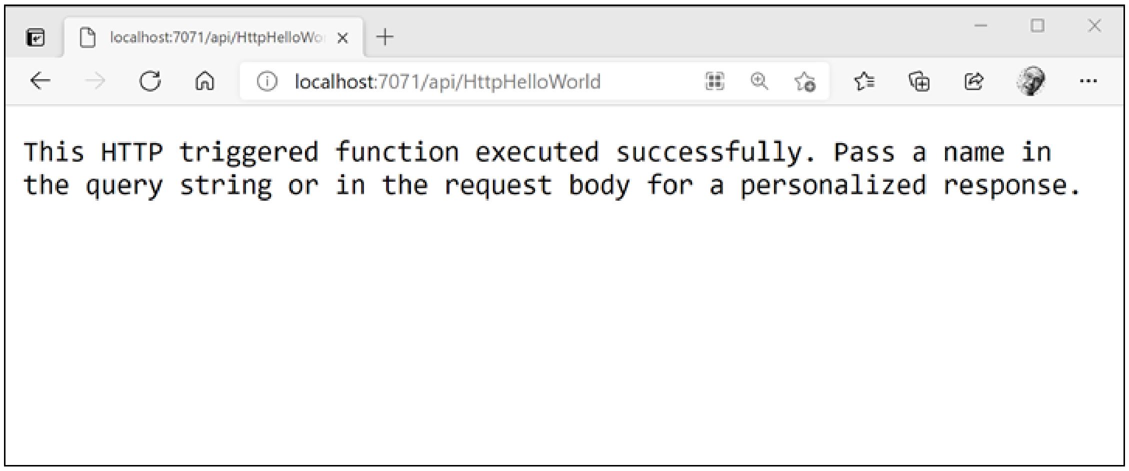
Using the previous code sample, the runtime exposes one HTTP-triggered function that can receive and manage **GET** and **POST** requests at http://localhost:7071/api/HttpHelloWorld.



```
PS C:\DurableFunctionsSuccynctly\MyFirstAzureFunction> func start
Microsoft (R) Build Engine version 16.9.0+57a23d249 for .NET
Copyright (C) Microsoft Corporation. All rights reserved.

  Determining projects to restore...
  All projects are up-to-date for restore.
  MyFirstAzureFunction -> C:\DurableFunctionsSuccynctly\MyFirstAzureFunction\bin\output\MyFirstAzureFunction.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:12.10


Azure Functions Core Tools
Core Tools Version:       3.0.3388 Commit hash: fb42a4e0b7fdc85fbd0bcfc8d743ff7d509122ae
Function Runtime Version: 3.0.15371.0

[2021-05-26T09:21:40.626Z] Found C:\DurableFunctionsSuccynctly\MyFirstAzureFunction\MyFirstAzureFunction.csproj. Usin
g for user secrets file configuration.

Functions:

        HttpHelloWorld: [GET,POST] http://localhost:7071/api/HttpHelloWorld

For detailed output, run func with --verbose flag.
```

*Figure 9: The output of the 'func start' command*

To test your function, you can open a browser and navigate to the URL exposed by the runtime. If you navigate to the URL http://localhost:7072/api/httpHelloWorld, you receive the response shown in the following figure.

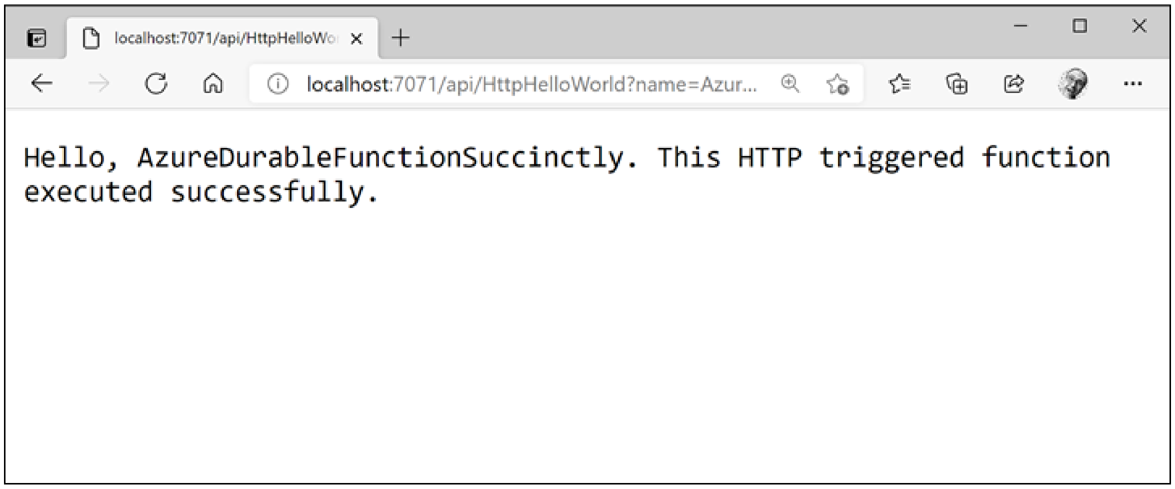


localhost:7071/api/HttpHelloWo
localhost:7071/api/HttpHelloWorld

```
This HTTP triggered function executed successfully. Pass a name in
the query string or in the request body for a personalized response.
```

*Figure 10: The response of the Azure function*

If you navigate to the URL http://localhost:7072/api/httpHelloWorld?name=AzureDurableFunctionSuccinctly, you receive a different response, shown in the following figure.


localhost:7071/api/HttpHelloWo
localhost:7071/api/HttpHelloWorld?name=Azur...

```
Hello, AzureDurableFunctionSuccinctly. This HTTP triggered function
executed successfully.
```

*Figure 11: The response of the Azure function*

In the previous function, we use only a trigger and a response binding. Now we try to add a new binding in the function. The scenario may be: if the HTTP request contains the parameter **`name`**, we respond with the message **hello, ...** (as we do in the function shown earlier), but at the same time, we write a message in a queue.

The function code is the following.

*Code Listing 2: The function with queue binding*

```
public static class HttpHelloWorld
{
    [FunctionName("HttpHelloWorld")]
    public static async Task<IActionResult> Run(
        [HttpTrigger(AuthorizationLevel.Function, "get", "post")]
HttpRequest req,
        [Queue("outputQueue",Connection ="storageConnection")]
IAsyncCollector<string> queue,
        ILogger log)
    {
        log.LogInformation("C# HTTP trigger function processed a
request.");

        string name = req.Query["name"];

        string requestBody = await new
StreamReader(req.Body).ReadToEndAsync();
        dynamic data = JsonConvert.DeserializeObject(requestBody);
        name = name ?? data?.name;

        if (!string.IsNullOrWhiteSpace(name))
        {
            await queue.AddAsync($"Request received for name {name}");
        }
        string responseMessage = string.IsNullOrEmpty(name)
            ? "This HTTP triggered function executed successfully. Pass a 
name in the query string or in the request body for a personalized 
response."
            : $"Hello, {name}. This HTTP triggered function executed 
successfully.";

        return new OkObjectResult(responseMessage);
    }
}
```

As you can see in the previous code, we add a new argument to the function method **`signature`**. The attribute **`Queue`** decorates the argument called **`queue`**. That is the declaration of the binding. We tell the runtime that we want to interact with a storage queue (the name of the storage queue is **`outputQueue`**, and it's configurable using the attribute's properties). The binding payload for that kind of binding is an instance of the **`IAsyncCollector`** interface.

***Note: The storage triggers and bindings are available in the package Microsoft.Azure.WebJobs.Extensions.Storage. You can add this package in your solution using NuGet integration provided by Visual Studio.***

If you rerun the function and navigate to the URL http://localhost:7072/api/httpHelloWorld?name=AzureDurableFunctionSuccinctly, you receive the same response you see in Figure 11, but the function should write a message in the queue.

One of the questions you might have looking at the code is: which queue will the message write? In other words, how can I configure the connection between my function and the storage containing the queue?

One of the arguments of the queue attribute is the **`Connection`** argument. You can use this argument to declare the configuration value (stored in the configuration file) to use as a connection string for the storage.

In the previous snippet of code, we set **`storageConnection`** as the value for the **`Connection`** argument. It means that the runtime tries to read the configuration file and retrieves the value for the key **`storageConnection`**.

To test the function locally, you can use the file **`local.settings.json`** , which should look like the following.

*Code Listing 3: The local.settings.json file*

```
{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "UseDevelopmentStorage=true",
    "FUNCTIONS_WORKER_RUNTIME": "dotnet",
    "storageConnection": "UseDevelopmentStorage=true"
  }
}
```

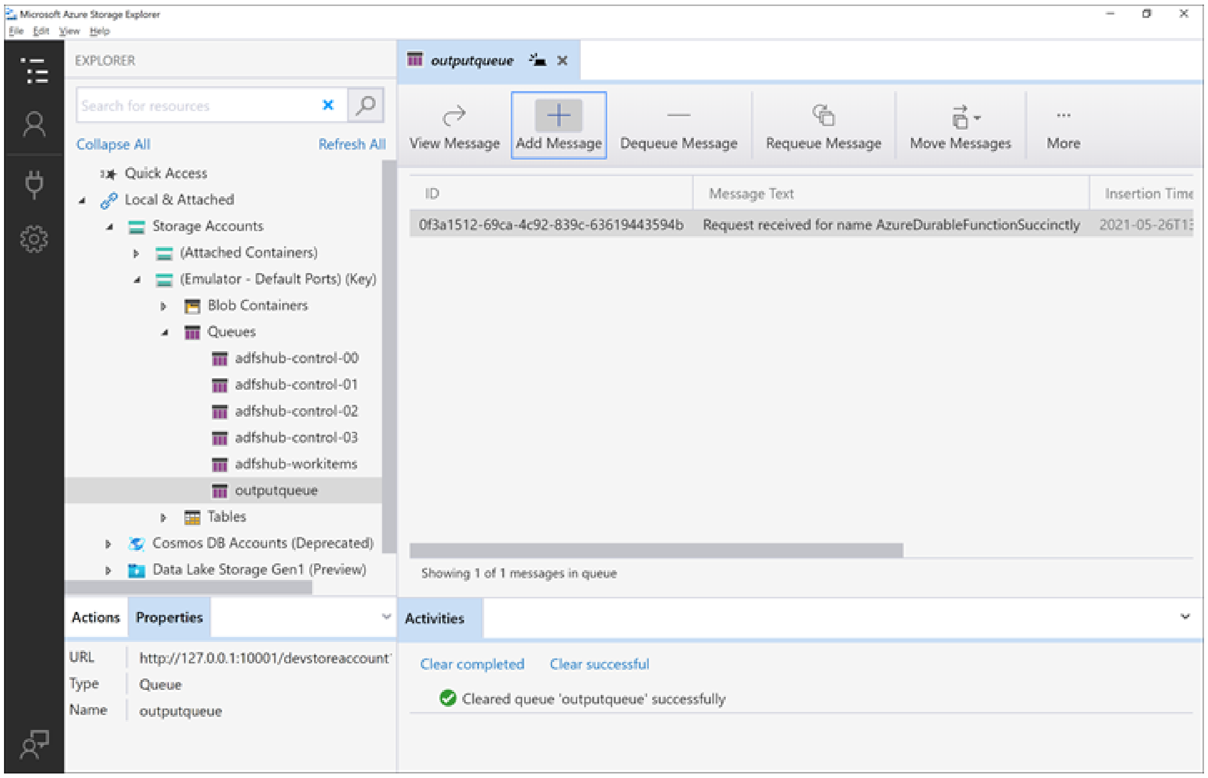
Because you are testing the functions locally, you can use the storage emulator on your machine to emulate the Azure storage account.  The connection string **`UseDevelopmentStorage=true`** tells the runtime to connect to the local emulator. Of course, when you deploy your functions on Azure, you will substitute the connection string with a valid connection string for a real storage account.

***Note: The storage emulator is installed automatically when you install the Microsoft Azure SDK, but if you want to install by yourself, you can download it here.***

If you want to verify that the function wrote the message in the queue, you can use Microsoft Azure Storage Explorer, a multiplatform tool you can use to interact with a storage account and the storage emulator.

***Note: You can download Microsoft Azure Storage Explorer here.***

Using the Storage Explorer tool, we can check if the function, during the execution, wrote the message in the queue. In the following figure, you can see the screenshot of the tools and the message related to the previous execution.



| ID | Message Text | Insertion Time |
|---|---|---|
| 0f3a1512-69ca-4c92-839c-63619443594b | Request received for name AzureDurableFunctionSuccinctly | 2021-05-26T1: |

Showing 1 of 1 messages in queue

| | |
|---|---|
| URL | http://127.0.0.1:10001/devstoreaccount |
| Type | Queue |
| Name | outputqueue |

Cleared queue 'outputqueue' successfully

*Figure 12: The Microsoft Azure Storage Explorer tool*

In this chapter, we don't go deep into the details about Azure Functions because its purpose is to recap the fundamentals of the technology.

An important thing to notice is that if you look at the previous snippet of code, both triggers and bindings are declared using attributes. You have a declarative approach, and the runtime is responsible for creating and managing the actual classes you use to get data from the trigger or interact with the external service through the binding.

In addition, those classes are POCO classes (POCO stands for plain old CLR objects, which means simple classes that contain data) or the implementation of interfaces. For those reasons, an Azure function is designed for unit testing. In fact, you can mock the trigger payload and the binding classes using your favorite testing framework. The runtime injects the actual classes for triggers and bindings into your Azure functions and manages the lifecycle for those objects.

***Note: All triggers and bindings, except for*** `TimerTrigger` ***and*** `HttpTrigger`***, are available using external packages. You can find those external packages in the NuGet portal.***

## Azure Functions extensibility

In the previous paragraph, we had a brief recap about Azure Functions, and the role and use of triggers and bindings.

The Azure Functions SDK is built on top of the Azure WebJob SDK, and it inherits the extensibility framework from that.

Even if you have 20 triggers and bindings provided by Microsoft, you can develop your own triggers or bindings leveraging the power of extensibility.

Understanding how the runtime and its binding process work is essential for using the extensibility model of the Azure functions. The binding process is the way the Azure Functions runtime creates and manages all the objects related to triggers and bindings.

In particular, the Azure Functions runtime has two different phases for the binding process:

- Startup binding: This phase occurs when the host of the functions starts. During this phase, the runtime registers all the extensions you add using packages and allows you to register your extensions. An extension is a particular class that contains your triggers and bindings definitions.

- Runtime binding: This phase occurs every time a trigger starts a function. During this phase, the runtime, using the binding classes of each trigger and binding involved in the process, creates the payloads and runs the function code.

An extension is a class that implements the **`IExtensionConfigProvider`** interface of the WebJob SDK. It is also called the binding provider, and it is registered in the runtime during the startup phase. It is responsible for creating all the infrastructural classes used by the runtime to manage the binding class for triggers and bindings.

The startup binding phase is composed of several steps executed by the runtime each time it starts to host your functions:

1. The runtime registers the integrated binding providers (for example, **`HttpTrigger`** and **`TimerTrigger`**) and allows you to register your binding providers (your extensions).
2. The runtime analyzes your code, finding all the methods you decorated with the attribute **`FunctionName`**.
3. For each method found in step 2 (potentially an Azure function), the runtime tries to resolve the dependency for each argument of the method using the binding classes defined in every single extension registered during step 1.
4. When all the Azure functions found in steps 2 and 3 are processed, the runtime creates an internal representation for each function. The runtime will use those representations during the runtime phase to execute the function quickly. In this step, the runtime excludes every function not resolved in step 3.
5. For each trigger used in each function, the runtime creates and executes the corresponding listener.

We will explain the role of the trigger listener later in this section.

The runtime binding phase is composed of two steps executed every time an Azure function is triggered:

1. The runtime retrieves the internal representation of the function (created in step 4 of the startup phase), and for each binding in the method signature, creates the binding objects the runtime uses to generates the actual classes injected in the execution.
2. If the runtime can generate all the binding objects correctly, then it executes the function passing them.

## Implement a custom trigger

The classes you need to implement your custom trigger are the following:

- **`TriggerConfigProvider`**: This class implements the **`IExtensionConfigProvider`** interface. In the **`Initialize`** method exposed by the **`IExtensionConfigProvider`** interface, you must define:
    - The attribute you use to decorate the function method.
    - The payload associated with the trigger (such as the **`HttpRequest`** in the **`HttpTrigger`**).
    - The binding provider class you use to create the actual binding object.
- **`TriggerAttribute`**: The developers use the attribute to decorate a method parameter to identify the function trigger. It inherits from **`Attribute`** class and is decorated by the **`BindingAttribute`**. It contains the properties you can use to configure the trigger behavior (such as the schedule expression for the **`TimerTrigger`**).
- **`TriggerBindingProvider`**: It implements the **`ITriggerBindingProvider`** interface and has the responsibility to create the actual binding object. It is a factory class in which you can read and validate the configuration data from the settings file.
- **`TriggerBinding`**: It's created by the **`TriggerBindingProvider`** and implements the **`ITriggerBinding`** interface. It creates the **`Listener`** instance and manages its lifecycle.
- **`TriggerListener`**: This class implements the **`IListener`** interface. It listens to the event source and reacts to the events to execute the function. There is an instance of the **`Listener`** for each function you define in your code.

**`TriggerBinding`** and **`Listener`** are important for the whole process, and you take care of their implementation. Performance and memory occupation of your functions depend on the implementation of those classes.

In the following figure, you can see how the aforementioned classes interact to implement the binding process.

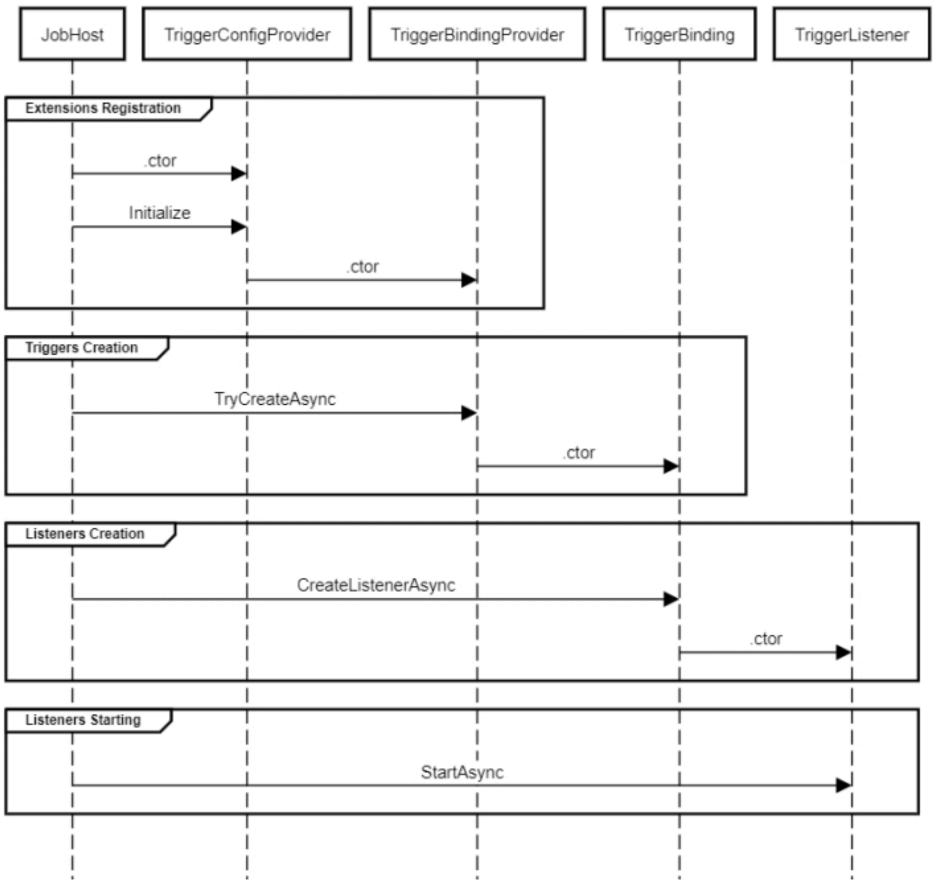


*Figure 13: Class interaction for the trigger binding process*

The entire process happens during the startup phase. It begins with the extension registration and ends with the listener startup.

***Note: You can find the implementation of BlobTrigger (provided by Microsoft) in the GitHub repository.***

## Implement a custom binding

The involved classes in the custom binding pipeline are the following:

- **`BindingAttribute`**: This attribute decorates a function parameter to identify the binding. It inherits from **`Attribute`** class and is decorated by the **`BindingAttribute`** (like the trigger attribute you saw in the previous chapter). It contains the binding configuration data (such as the queue name for a **QueueBinding**) for binding in the function in which you use it. It is used by developers to declare they want to use your binding.

- **BindingConfigProvider**: It implements the **IExtensionConfigProvider** interface and adds the extensions to the runtime. A binding is defined using one or more binding rules. Defining a rule means that you:
    - Declare what kind of binding attribute identifies the rule. When the runtime discovers that a binding attribute is used in a function, it then uses the corresponding rule.
    - Add, if you want, a validator for the attribute (for example, you can check that a connection string is formally valid).
    - Add the behavior for the rule that is the type of binding you want to support. It depends on what kind of object you want to use to implement the binding feature. You can have three types of bindings:
        - **BindToInput**: You declare that your binding type is an object. You use this kind of behavior when you want to bind your data to a single input (such as a single row in a **CloudTable**).
        - **BindToCollector**: You declare that you support a list in your binding, meaning your binding object is a collection that implements **IAsyncCollector** interface (for example, if you want to add multiple items to a queue).
        - **BindToStream**: You want to support a stream as binding.
        - **BindingConverter**: This is the class that creates the actual binding object. It must implement the **IConverter** interface. The runtime uses this class every time it runs a function to create the binding object.
        - **BindingInstance**: The **Binding** class is the class that allows the developer to interact with the external service.

The following figure shows the interactions between the classes mentioned before in the different phases of the binding process.

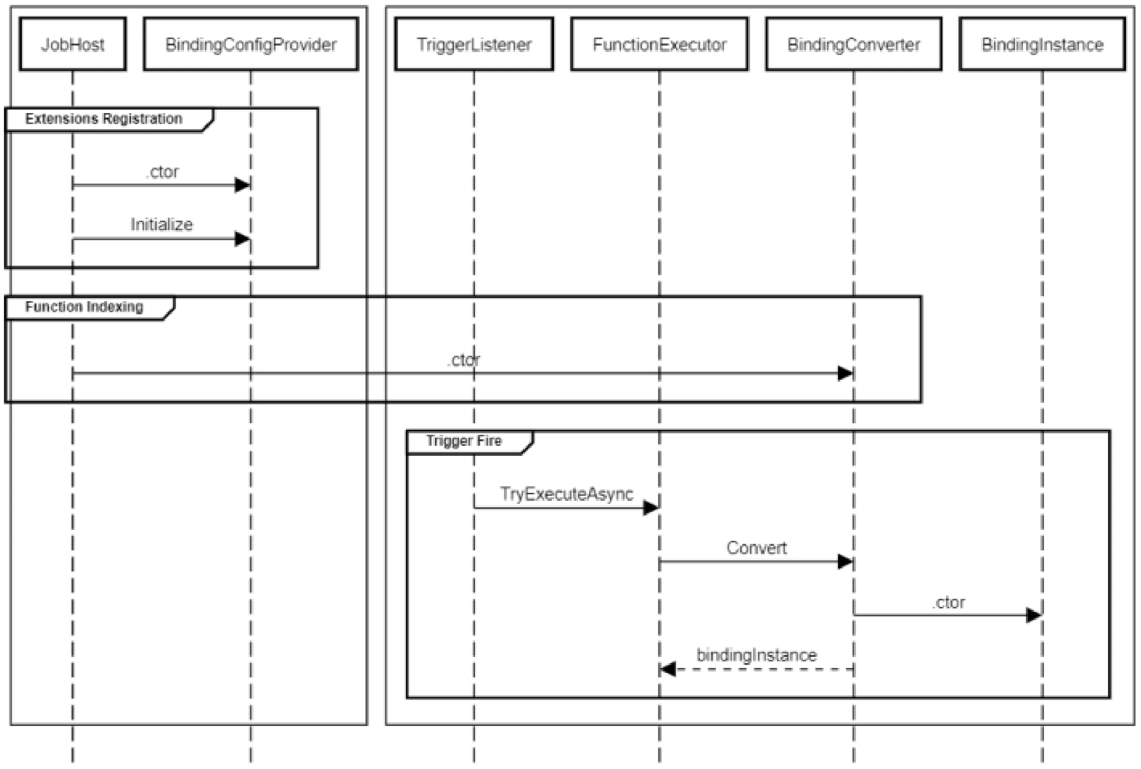


*Figure 14: Classes interaction for the binding process*

During the startup phase of the functions host, the runtime uses your binding config provider to register the extension and creates an instance of the **`BindingConverter`** class to store the internal representation of every single function. During the runtime phase, when a listener runs a function that involves the custom binding, the runtime (in particular the **`FunctionExecutor`**, the runtime component that manages the execution context for the functions) uses the **`BindingConverter`** created in the startup phase to generate the binding instance and passes it to the function running.

You need to take care of the **`BindingConverter`** class implementation, because it manages the lifecycle of the classes that every function uses to interact with external services.

***Note: You can find the implementation of the binding for CosmosDB (provided by Microsoft) in the GitHub repository.***

# Chapter 2 Durable Functions Fundamentals

## Durable Functions components

Azure Functions is a great technology to implement using your favorite programming language, a unit of software. You can debug and test the functions easily, but they have a few limitations.

First, you cannot call a function directly from another function. Of course, you can make an HTTP call or use a queue, but you cannot call the other function directly, and this limitation makes your solution more complex.

The second (and more critical) limitation is that you cannot create a stateful function, which means you cannot store data in the function. Of course, you can use external services like storage or CosmosDB to store your data, but you have to implement the persistence layer in your function, and again, this situation makes your solution more complex.

For example, if you want to create a workflow using Azure Functions, you have to store the workflow status and interact with other functions.

A workflow is a set of stages, and each of them is typically a function call. You have to store the stage you reach and call other functions directly. You can implement a workflow using Azure Functions, but you need to find a solution to store (using an external service and some bindings) the workflow status and call the functions that compose your workflow.

Durable Functions is an Azure Functions extension that implements triggers and bindings that abstract and manage state persistence. Using Durable Functions, you can easily create stateful objects entirely managed by the extension.

You can implement Durable Functions using C#, F#, JavaScript, Python, and PowerShell.

***Note: If you implement your Durable Functions using C#, you need to reference the package Microsoft.Azure.WebJobs.Extensions.DurableTask.***

### Clients, orchestrators, and activities

In the following figure, you can see the main components in a Durable function.

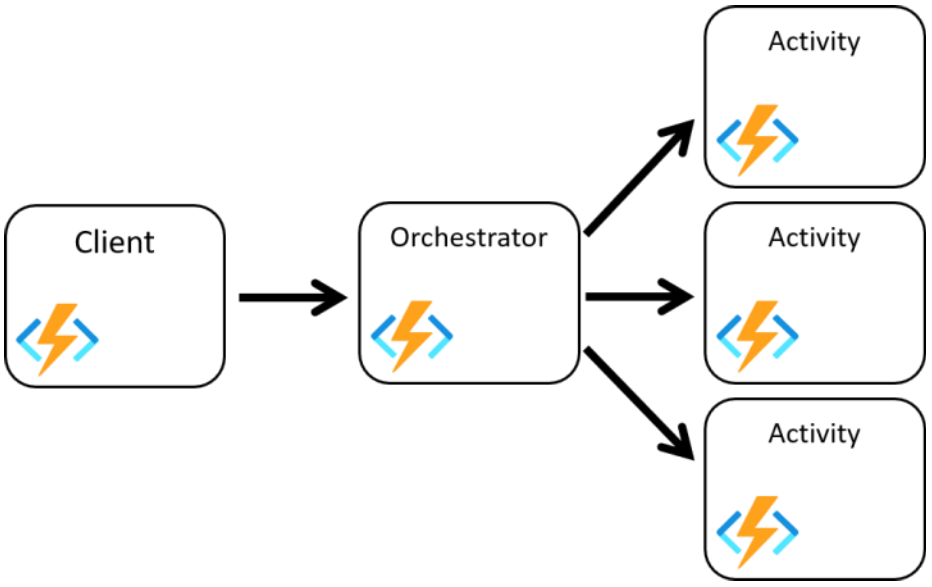


*Figure 15: Durable Functions components*

All the components in the Durable Functions world are Azure functions, but they use specific triggers and bindings provided by the extension.

Every component has its responsibility:

- **Client**: The client starts or sends commands to an orchestrator. It is an Azure function triggered by a built-in (or custom) trigger as a standard Azure function. It uses a custom binding (**`DurableClient`**, provided by the Durable Functions extension) to interact with the orchestrator.
- **Orchestrator**: The orchestrator implements your workflow. Using code, it describes what actions you want to execute and the order of them. An orchestrator is triggered by a particular trigger provided by the extension (**`OrchestrationTrigger`**). The **`OrchestrationTrigger`** gives you a context of execution you must use to call activities or other orchestrators. The purpose of the orchestrator is only to orchestrate activities or other orchestrators (sub-orchestration). It doesn't have to perform any calculation or I/O operations, but only governs activities' flow.
- **Activity**: The basic unit of work of your workflow, the activity is an Azure function triggered by an orchestrator using the **`ActivityTrigger`** provided by the extension. The activity trigger gives you the capability to receive data from the orchestrator. Inside the activity, you can use all the bindings you need to interact with external systems.

You have another type of function in the Durable Functions platform called the entity, but we will see it in another chapter later in this book.

The triggers and bindings provided by the extension abstract the state persistence.

## How Durable functions manage state: function chaining

The Durable Functions extension is based on the Durable Task Framework. The Durable Task Framework is an open-source library provided by Microsoft that helps you implement long-running orchestration tasks. It abstracts the workflow state persistence and manages the orchestration restart and replay.

***Note: The Durable Task Framework is available on GitHub.***

The Durable Task Framework uses Azure Storage as the default persistence store for the workflow state, but you can use different stores using extended modules. At the moment, you can use Service Bus, SQL Server, or Netherite. To configure a different persistence store, you need to change the **`host.json`** file.

***Note: Netherite is a workflow execution engine based on Azure EventHub and FASTER (a high-performance key-value store), implemented by Microsoft Research.***

We will refer to the default persistence layer (Azure Storage) during the paragraph's continuation, but what we will say is almost the same for any other persistence layer.

Durable functions use queues, tables, and blobs inside the storage account you configure in your function app settings for the following purposes:

- **Queues**: The persistence layer uses the queue to schedule orchestrators and activities' executions. For example, every time the platform needs to invoke an activity from an orchestrator, it creates a specific message in one of the queues. The **`ActivityTrigger`** reacts to that message and starts the right activity.
- **Tables**: Durable functions use storage tables to store the status of each orchestrator (instances table) and all their execution events (history table). Every time an orchestrator does something (such as call an activity or receive a result from an activity), the platform writes one or more events in the history table. This event sourcing approach allows the platform to reconstruct the actual status of each orchestrator during every restart.
- **Blobs**: Durable functions use the blobs when the platform needs to save data with a size greater than the limitations of queues and tables (regarding the maximum size of a single entity or a single message).

The Durable Functions platform uses JSON as standard, but you can customize the serialization by implementing your custom serialization component.

Queues, tables, and blobs are grouped in a logical container called the task hub. You can configure the name of the task hub for each function app by modifying the **`host.json`** file. The following file is an example.

*Code Listing 4: Setting the name of the task hub in the host.json file*

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

Using the previous **`host.json`** file, the Durable Functions platform will create the queues, table, and blobs shown in the following figure.

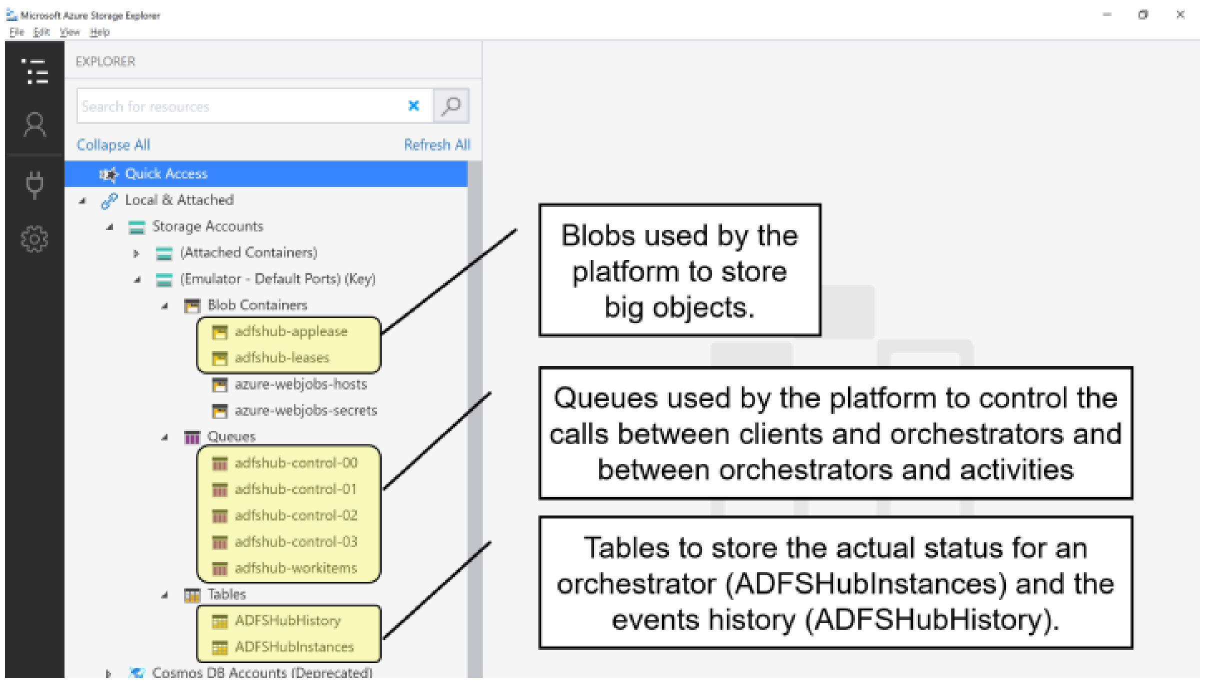

Microsoft Azure Storage Explorer
File Edit View Help
EXPLORER
Search for resources
Collapse All
Refresh All
Quick Access
Local & Attached
Storage Accounts
(Attached Containers)
(Emulator - Default Ports) (Key)
Blob Containers
adfshub-applease
adfshub-leases
azure-webjobs-hosts
azure-webjobs-secrets
Queues
adfshub-control-00
adfshub-control-01
adfshub-control-02
adfshub-control-03
adfshub-workitems
Tables
ADFSHubHistory
ADFSHubInstances
Cosmos DB Accounts (Deprecated)
Blobs used by the platform to store big objects.
Queues used by the platform to control the calls between clients and orchestrators and between orchestrators and activities
Tables to store the actual status for an orchestrator (ADFSHubInstances) and the events history (ADFSHubHistory).

*Figure 16: Queues, tables, and blobs in a task hub*

As you can see, the name you configure in the **`hubName`** property of the **`host.json`** file becomes the prefix of all the queues, tables, and blobs used by the platform.

To better explain the mechanism used by Durable Functions to store the progress of an orchestrator, let's consider a simple example of a workflow called function chaining.

In the following figure, you can see a simple function chaining.

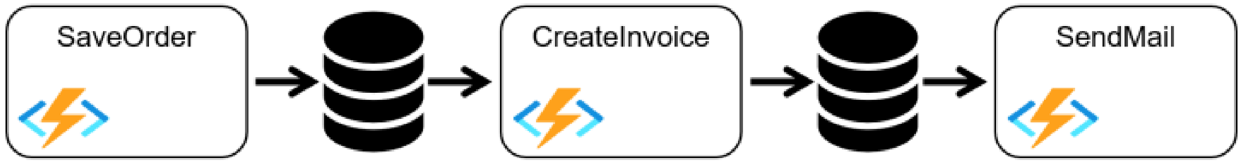


*Figure 17: Function chaining*

In this diagram, we suppose a simple workflow that manages a customer order. A client function receives an HTTP request that contains the order. The client starts an orchestrator, and the orchestrator calls the following activities:

- **`SaveOrder`**: Receives the order from the orchestrator and saves it in table storage (the orders table).
- **`CreateInvoice`**: Creates an invoice file in blob storage.
- **`SendMail`**: Sends an email to the customer for order confirmation.

Before looking at the code for the client, the orchestrator, and the activities, you must notice that a pattern like this can be implemented using standard Azure Functions and storage queues.

Each function could call the next one in the sequence using a queue, but you must manage two different queues, and if the number of activities grows, the number of queues becomes too high. Furthermore, the relationship between each function and its queue becomes an implementation detail, and you must document it very well.

Finally, your solution can become very complex to manage if you have to introduce compensation mechanisms in case of errors. Imagine you want to call a different function if the **`CreateInvoice`** activity throws an exception. How can you do this? Of course, you can introduce different queues for each exception you want to manage, but you understand that the complexity of your implementation just became unmanageable.

***Note: You can find all the code used in this book at [https://github.com/massimobonanni/AzureDurableFunctionsSuccinctly](https://github.com/massimobonanni/AzureDurableFunctionsSuccinctly).***

Now we can take a look at the code starting from the client.

*Code Listing 5: Function chaining client*

```
[FunctionName("OrderManager_Client")]
public async Task<HttpResponseMessage> Client(
    [HttpTrigger(AuthorizationLevel.Function, "post", Route =
"functionchaining/order")] HttpRequestMessage req,
    [DurableClient] IDurableOrchestrationClient starter,
    ILogger log)
{
    log.LogInformation($"[CLIENT OrderManager_Client] --> Order
received!");
    string jsonContent = await req.Content.ReadAsStringAsync();

    try
    {
        var order = JsonConvert.DeserializeObject<Order>(jsonContent);
        var instanceId = await
starter.StartNewAsync("OrderManager_Orchestrator", order);

        log.LogInformation($"Order received - started orchestration with
ID = '{instanceId}'.");

        return starter.CreateCheckStatusResponse(req, instanceId);
    }
    catch (Exception ex)
    {
        log.LogError("Error during order received operation", ex);
    }

    return new HttpResponseMessage(System.Net.HttpStatusCode.BadRequest);
}
```

The client is an Azure function with an HTTP trigger and uses the **`DurableClient`** attribute to declare the binding that allows you to manage the orchestrator.

The **`IDurableOrchestrationClient`** instance provided by the Durable Functions platform is the payload class for the new binding. **`StartNewAsync`** allows you to start a new orchestration, giving the name of the orchestration function and a custom object (in our sample, the order deserialized from the request) and returns the instance ID of the orchestrator.

Every time you start a new orchestrator, the platform creates a new instance and provides you with an ID that allows you to manage and control the specific orchestration.

The client function finishes returning a response generated by the method **`CreateCheckStatusResponse`**. We will talk about the content of this response in the paragraph related to the async HTTP APIs pattern.

*Code Listing 6: Function chaining orchestrator*

```
[FunctionName("OrderManager_Orchestrator")]
public async Task<Invoice> Orchestrator(
    [OrchestrationTrigger] IDurableOrchestrationContext context,
    ILogger log)
{
    log.LogInformation($"[ORCHESTRATOR OrderManager_Orchestrator] --> id
: {context.InstanceId}");

    var order = context.GetInput<Order>();

    var orderRow = await
context.CallActivityAsync<OrderRow>("OrderManager_SaveOrder", order);

    var invoice = await
context.CallActivityAsync<Invoice>("OrderManager_CreateInvoice",
orderRow);

    await context.CallActivityAsync("OrderManager_SendMail", invoice);

    return invoice;
}
```

The previous code contains the implementation for the orchestrator for our function chaining pattern.

The **OrchestrationTrigger** trigger starts the function and manages the restart every time an activity, called by the orchestrator, finishes. You can also use the trigger to retrieve the object passed by the client (using the **GetInput** method).

The orchestrator must implement only the workflow flow, and must not implement calculation, I/O operation, or similar. The code you write in an orchestrator must be deterministic. This means you cannot use values generated inside the orchestrator (like a random value), or values can change every time the orchestrator runs (for example, a value related to local time). Remember that the platform restarts the orchestrator every time an activity finishes and rebuilds its state using the events stored in the history table mentioned before. The state-building phase must be deterministic; otherwise, you receive an error.

If you need a random value, for example, you can create an activity to generate it and call the activity from the orchestrator.

As you can see in the previous code, you can call an activity (using the method **CallActivityAsync**), passing an object, and retrieve the result. The objects you involved in the process are custom objects (so you can create the classes you need to pass the data you need), and the only limitation is that those classes must be JSON serializable.

Now complete the code showing the three activities.

*Code Listing 7: Function chaining save order activity*

```
[FunctionName("OrderManager_SaveOrder")]
public async Task<OrderRow> SaveOrder([ActivityTrigger] Order order,
    [Table("ordersTable", Connection = "StorageAccount")]
IAsyncCollector<OrderRow> ordersTable,
    ILogger log)
{
    log.LogInformation($"[ACTIVITY OrderManager_SaveOrder] --> order :
{order}");

    var orderRow = new OrderRow(order);
    await ordersTable.AddAsync(orderRow);
    return orderRow;
}
```

The **SaveOrder** activity is an Azure function triggered by an **ActivityTrigger**. The **ActivityTrigger**, like the **OrchestrationTrigger**, uses the queues mentioned earlier to schedule the execution of the function when an orchestrator requests it using the **CallActivityAsync** method.

This method creates a message containing all the information the platform needs to start the activity and puts it in the **TaskHub-workitems** queue, as shown in Figure 16. The **ActivityTrigger** is listening to that queue and can start the activity. The following JSON shows an example of a message created by the orchestrator.

*Code Listing 8: An example of a message generated by the orchestrator*

```
{
  "$type": "DurableTask.AzureStorage.MessageData",
  "ActivityId": "3a718fb6-80fc-4a90-ba92-6366da789f1d",
  "TaskMessage": {
    "$type": "DurableTask.Core.TaskMessage",
    "Event": {
      "$type": "DurableTask.Core.History.TaskScheduledEvent",
      "EventType": 4,
      "Name": "OrderManager_SaveOrder",
      "Version": "",
      "Input": "[{\"$type\":\"DurableFunctions.FunctionChaining.Order,
DurableFunctions\",\"custName\":\"Massimo Bonanni\",\"custAddress\":\"Viale
Giuseppe Verdi 1,
Roma\",\"custEmail\":\"massimo.bonanni@microsoft.com\",\"cartId\":\"1111111
1\",\"date\":\"2021-05-18T13:00:00\",\"price\":435.0,\"fileName\":null}]",
      "EventId": 0,
      "IsPlayed": false,
      "Timestamp": "2021-06-07T14:26:30.4134592Z"
    },
    "SequenceNumber": 0,
    "OrchestrationInstance": {
      "$type": "DurableTask.Core.OrchestrationInstance",
      "InstanceId": "de534343c593443ca45dbc6165eb44ce",
      "ExecutionId": "b064f9960f0447e983547c8d0801d5b6"
    }
  },
  "CompressedBlobName": null,
  "SequenceNumber": 2,
  "Episode": 1,
  "Sender": {
    "InstanceId": "de534343c593443ca45dbc6165eb44ce",
    "ExecutionId": "b064f9960f0447e983547c8d0801d5b6"
  },
  "SerializableTraceContext": null
}
```

The message contains all the information the platform needs to start the right activity:

- **`Input`**: The object (serialized) passed from the orchestrator to the activity.
- **`Name`**: The name of the activity to call.
- **`OrchestrationInstance`**: The orchestration information in terms of **`InstanceId`** and **`ExecutionId`**.

In the following snippets of code, you can see the implementation of the other two activities involved in the workflow.

*Code Listing 9: Function chaining create invoice activity*

```
[FunctionName("OrderManager_CreateInvoice")]
[StorageAccount("StorageAccount")]
public async Task<Invoice> CreateInvoice([ActivityTrigger] OrderRow
order,
    IBinder outputBinder,
    ILogger log)
{
    log.LogInformation($"[ACTIVITY OrderManager_CreateInvoice] --> order
: {order.orderId}");

    var fileName = $"invoices/{order.orderId}.txt";

    using (var outputBlob = outputBinder.Bind<TextWriter>(new
BlobAttribute(fileName)))
    {
        await outputBlob.WriteInvoiceAsync(order);
    }

    var invoice = new Invoice() { order = order, fullPath = fileName };

    return invoice;
}
```

The **CreateInvoice** activity receives the order written in the storage table by the **SaveOrder** activity and creates the order invoice in a blob storage container.

*Code Listing 10: Function chaining send mail activity*

```
[FunctionName("OrderManager_SendMail")]
[StorageAccount("StorageAccount")]
public async Task SendMail([ActivityTrigger] Invoice invoice,
     [SendGrid(ApiKey = "SendGridApiKey")]
IAsyncCollector<SendGridMessage> messageCollector,
     IBinder invoiceBinder,
     ILogger log)
{
    log.LogInformation($"[ACTIVITY OrderManager_SendMail] --> invoice :
{invoice}");

    SendGridMessage message;
    using (var inputBlob = invoiceBinder.Bind<TextReader>(new
BlobAttribute(invoice.fileName)))
    {
        message = await SendGridHelper.CreateMessageAsync(invoice,
inputBlob);
    }
    await messageCollector.AddAsync(message);
}
```

Finally, the **`SendMail`** activity receives the invoice data from the orchestrator and sends an email to the customer using the **`SendGrid`** binding provided by Microsoft.

> ***Note: [SendGrid]() is an email delivery service that allows you to automate the email sending process using REST APIs. Microsoft provides a set of bindings to interact with SendGrid, and you can find them in the package [Microsoft.Azure.WebJobs.Extensions.SendGrid](). You can find more information on how you can use SendGrid in your Azure functions [here]().***

## Orchestrator event-sourcing pattern

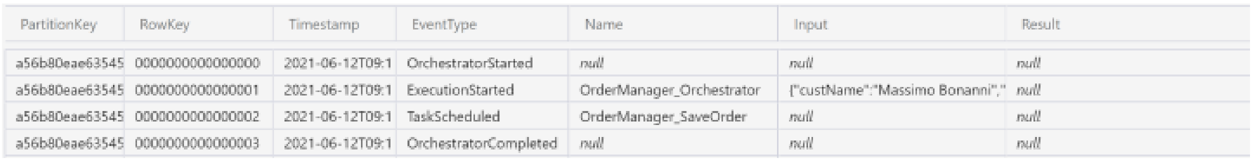
We previously discussed function chaining, one of the simplest workflow patterns we can implement using Durable Functions. The question is: how does the platform execute every single instance of the orchestrator, reconstructing precisely the state it had reached?

Durable functions are Azure functions, and therefore, cannot have a long execution time. The Durable Functions platform runs the orchestrator for the time necessary to perform the steps required to invoke one of the activities. It stops (literally) and restarts when the activity completes its job. At that point, the platform must reconstruct the state the orchestrator had reached to resume the execution with the remaining activities.

The platform manages the state of every single orchestrator using the event-sourcing pattern. Every time the orchestrator needs to start an activity, the platform stops its execution and writes a set of rows in a storage table. Each row contains what the orchestrator does before starting the activity (events sourcing).

As an example, let's imagine we are running the orchestrator shown in Code Listing 6. The platform executes the instructions until the orchestrator calls the **`SaveOrder`** activity. The platform writes the startup message of the activity in the queue (as we saw previously), stops the execution of the orchestrator, and writes a few rows in the **`TaskHubHistory`** table to store the progress of the workflow.

In the next figure, you can see an example of what the platform writes in the table.



| PartitionKey | RowKey | Timestamp | EventType | Name | Input | Result |
|---|---|---|---|---|---|---|
| a56b80eae63545 | 0000000000000000 | 2021-06-12T09:1 | OrchestratorStarted | *null* | *null* | *null* |
| a56b80eae63545 | 0000000000000001 | 2021-06-12T09:1 | ExecutionStarted | OrderManager_Orchestrator | {"custName":"Massimo Bonanni"," | *null* |
| a56b80eae63545 | 0000000000000002 | 2021-06-12T09:1 | TaskScheduled | OrderManager_SaveOrder | *null* | *null* |
| a56b80eae63545 | 0000000000000003 | 2021-06-12T09:1 | OrchestratorCompleted | *null* | *null* | *null* |

*Figure 18: Orchestrator history before calling SaveOrder activity*

This table shows only a subset of all the columns used in the history table. We analyze only the columns helpful to understand the event-sourcing pattern.

The meaning of each column is the following:

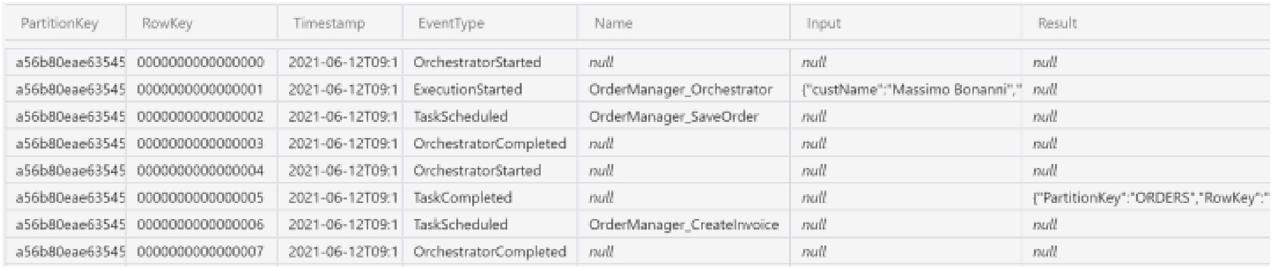
- **`PartitionKey`**: The platform uses the orchestrator instance ID to partition events (and therefore, the state). The events of each instance of the orchestrator will always be in the same partition.
- **`RowKey`**: The platform uses the order of succession of events as a row key.
- **`TimeStamp`**: This is the time in which the event occurred.
- **`EventType`**: This is the event type. For example, the first event is always **`OrchestratorStarted`**. The event type **`TaskScheduled`** is used to signal that the orchestrator starts an activity.
- **`Name`**: If the field is used, it contains the name of the functions object of the event. For example, if you look at RowKey 1, you can notice that the name field contains the value **`OrderManager_Orchestrator`**, which is exactly the name of the orchestrator function. In the case of an activity, we will find the name of the activity (look at RowKey 2).
- **`Input`**: This field contains the serialization of all the input objects passed to the orchestrator. In RowKey 1, you can see the payload received by the orchestrator from the client function.
- **`Result`**: This contains the serialization of the objects returned by the single activity when they finish their job.

When the **`SaveOrder`** activity finishes its job, the platform writes a new message in the communication queue to start the orchestrator again. The **`OrchestrationTrigger`** captures that message and starts the orchestrator function (the message contains the ID of the instance of the orchestrator, so the platform can start the correct orchestrator instance).

The platform reads the rows in the history table and reconstructs the workflow state. It knows that the orchestrator already called the **`SaveOrder`** activity, and it must proceed with the instructions after the **`CallActivityAsync`** method.

Again, the platform creates a message in the communication queue to call the **`CreateInvoice`** function, stops the orchestrator, and writes some rows in the history table.

The new situation for the history table is shown in the following figure.



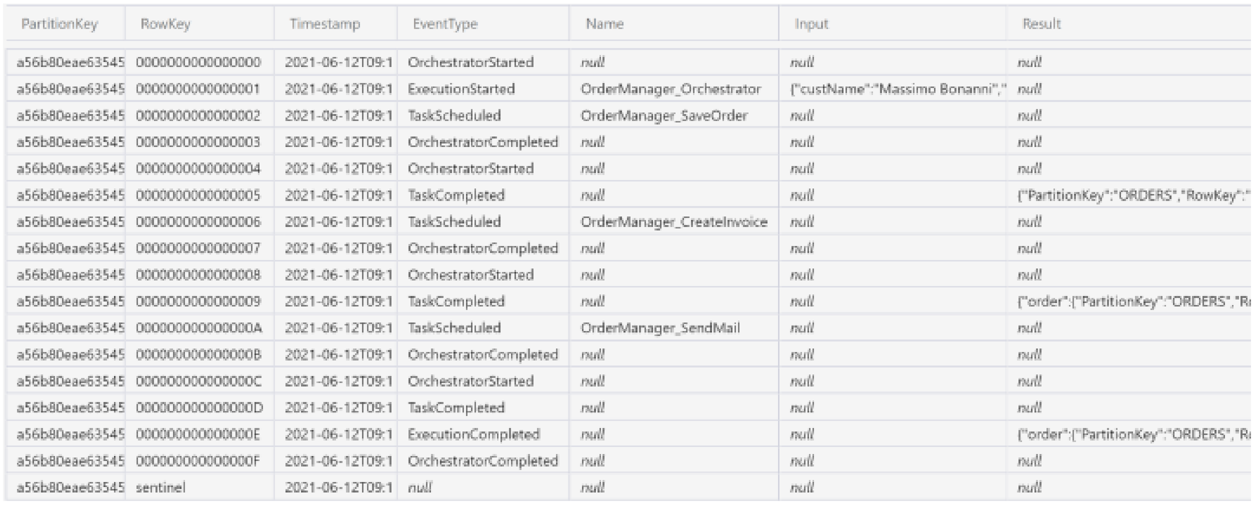
| PartitionKey | RowKey | Timestamp | EventType | Name | Input | Result |
|---|---|---|---|---|---|---|
| a56b80eae63545 | 0000000000000000 | 2021-06-12T09:1 | OrchestratorStarted | *null* | *null* | *null* |
| a56b80eae63545 | 0000000000000001 | 2021-06-12T09:1 | ExecutionStarted | OrderManager_Orchestrator | {"custName":"Massimo Bonanni"," | *null* |
| a56b80eae63545 | 0000000000000002 | 2021-06-12T09:1 | TaskScheduled | OrderManager_SaveOrder | *null* | *null* |
| a56b80eae63545 | 0000000000000003 | 2021-06-12T09:1 | OrchestratorCompleted | *null* | *null* | *null* |
| a56b80eae63545 | 0000000000000004 | 2021-06-12T09:1 | OrchestratorStarted | *null* | *null* | *null* |
| a56b80eae63545 | 0000000000000005 | 2021-06-12T09:1 | TaskCompleted | *null* | *null* | {"PartitionKey":"ORDERS","RowKey":" |
| a56b80eae63545 | 0000000000000006 | 2021-06-12T09:1 | TaskScheduled | OrderManager_CreateInvoice | *null* | *null* |
| a56b80eae63545 | 0000000000000007 | 2021-06-12T09:1 | OrchestratorCompleted | *null* | *null* | *null* |

*Figure 19: Orchestrator history before calling the CreateInvoice activity*

You can see the serialization of the return object provided by the save order activity in the result field of RowKey 0005.

The platform iterates the pattern until the orchestrator finishes its execution. Every time the orchestrator restarts because an activity finishes its job, the platform reconstructs the state using the history table.

You can see all the history rows written by the platform for a function chaining orchestrator in the following figure.

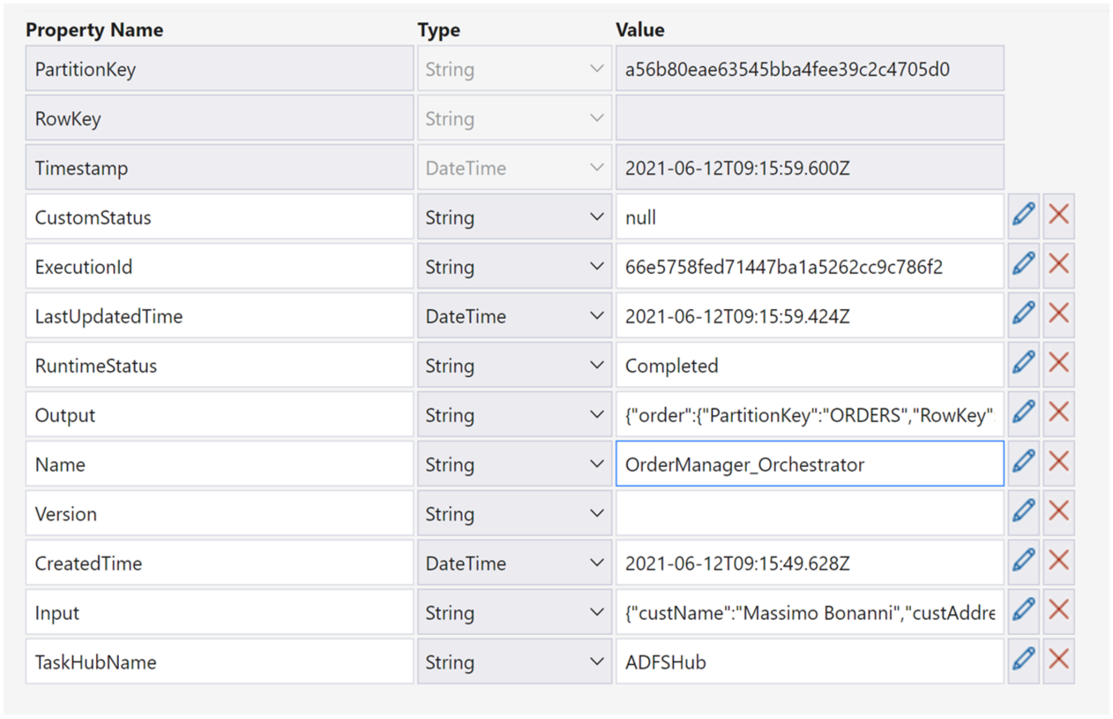


| PartitionKey | RowKey | Timestamp | EventType | Name | Input | Result |
|---|---|---|---|---|---|---|
| a56b80eae63545 | 0000000000000000 | 2021-06-12T09:1 | OrchestratorStarted | *null* | *null* | *null* |
| a56b80eae63545 | 0000000000000001 | 2021-06-12T09:1 | ExecutionStarted | OrderManager_Orchestrator | {"custName":"Massimo Bonanni"," | *null* |
| a56b80eae63545 | 0000000000000002 | 2021-06-12T09:1 | TaskScheduled | OrderManager_SaveOrder | *null* | *null* |
| a56b80eae63545 | 0000000000000003 | 2021-06-12T09:1 | OrchestratorCompleted | *null* | *null* | *null* |
| a56b80eae63545 | 0000000000000004 | 2021-06-12T09:1 | OrchestratorStarted | *null* | *null* | *null* |
| a56b80eae63545 | 0000000000000005 | 2021-06-12T09:1 | TaskCompleted | *null* | *null* | {"PartitionKey":"ORDERS","RowKey":" |
| a56b80eae63545 | 0000000000000006 | 2021-06-12T09:1 | TaskScheduled | OrderManager_CreateInvoice | *null* | *null* |
| a56b80eae63545 | 0000000000000007 | 2021-06-12T09:1 | OrchestratorCompleted | *null* | *null* | *null* |
| a56b80eae63545 | 0000000000000008 | 2021-06-12T09:1 | OrchestratorStarted | *null* | *null* | *null* |
| a56b80eae63545 | 0000000000000009 | 2021-06-12T09:1 | TaskCompleted | *null* | *null* | {"order":{"PartitionKey":"ORDERS","R |
| a56b80eae63545 | 000000000000000A | 2021-06-12T09:1 | TaskScheduled | OrderManager_SendMail | *null* | *null* |
| a56b80eae63545 | 000000000000000B | 2021-06-12T09:1 | OrchestratorCompleted | *null* | *null* | *null* |
| a56b80eae63545 | 000000000000000C | 2021-06-12T09:1 | OrchestratorStarted | *null* | *null* | *null* |
| a56b80eae63545 | 000000000000000D | 2021-06-12T09:1 | TaskCompleted | *null* | *null* | *null* |
| a56b80eae63545 | 000000000000000E | 2021-06-12T09:1 | ExecutionCompleted | *null* | *null* | {"order":{"PartitionKey":"ORDERS","R |
| a56b80eae63545 | 000000000000000F | 2021-06-12T09:1 | OrchestratorCompleted | *null* | *null* | *null* |
| a56b80eae63545 | sentinel | 2021-06-12T09:1 | *null* | *null* | *null* | *null* |

*Figure 20: Full execution history for the function chaining orchestrator*

Another helpful table used by the platform is the **`TaskHubInstances`** table. The platform creates one row for each orchestrator instance in this table, and each row contains the current status of the orchestrator.

In the following figure, you can see the instance row for the previous orchestrator execution.



| Property Name | Type | Value |
|---|---|---|
| PartitionKey | String | a56b80eae63545bba4fee39c2c4705d0 |
| RowKey | String | |
| Timestamp | DateTime | 2021-06-12T09:15:59.600Z |
| CustomStatus | String | null |
| ExecutionId | String | 66e5758fed71447ba1a5262cc9c786f2 |
| LastUpdatedTime | DateTime | 2021-06-12T09:15:59.424Z |
| RuntimeStatus | String | Completed |
| Output | String | {"order":{"PartitionKey":"ORDERS","RowKey" |
| Name | String | OrderManager_Orchestrator |
| Version | String | |
| CreatedTime | DateTime | 2021-06-12T09:15:49.628Z |
| Input | String | {"custName":"Massimo Bonanni","custAddre |
| TaskHubName | String | ADFSHub |

*Figure 21: The actual state of an orchestrator in the instances table*

As you can see in the table in Figure 21, the platform saves all the information you can use to understand the current state of each orchestrator. The most interesting fields are:

- **`LastUpdatedTime`**: This is the timestamp of the last update.
- **`RuntimeStatus`**: This field contains the current status of the orchestrator and allows you to understand if the orchestrator finishes its job.
- **`Output`**: In this field, you can find the return object of the orchestrator (if the orchestrator returns an object). The returned object is serialized.
- **`Input`**: Here, you can find the input object passed to the orchestrator. Also, in this case, the object is serialized.

## Versioning issues

The event sourcing approach allows the platform to understand the step the orchestrator reached in the previous execution and reconstruct the state. But what happens if we change code in the orchestrator (or in one of the activities used by it)?

Of course, there aren't issues with the new executions: they run using the new code. You can have problems with the orchestrator in the running state (the orchestrator instances that have started their job but not finished it yet). After code changes, in fact, at the next execution, the platform tries to reconstruct the orchestrator status—but the code in the orchestrator could be different from the one that generated the set of events stored in the history table. In that scenario, the orchestrator instance throws an exception.

You need to take care of those breaking changes. Two of the most common are:

- **Change orchestrator or activity signature**: This breaking change occurs when you modify the number or types of the arguments of an orchestrator or activity.
- **Change the orchestration logic**: In this breaking change, you change the code inside an orchestrator function, and the new one is different from the previous in terms of workflow.

There are some strategies to manage those breaking changes:

- **Do nothing**: Using this approach, you continue to receive errors from the orchestrators (because the platform tries to run the orchestrator every time an exception is generated). It isn't the recommended solution.
- **Stop all running instances**: You can stop all the running instances by removing the rows in the management tables for those orchestrators (if you have access to the underlying storage resources). Alternatively, you can restart all the orchestrators using the features exposed by the platform. We will see those features in the next chapters.
- **Deploy the new version side-by-side with the previous one**: In this scenario, you can use different storage accounts for different function versions, keeping each version isolated from the other.

If you are deploying your functions in Azure, you can use deployment slots provided by the Azure Functions platform to deploy multiple versions of your functions in isolated environments.

# Chapter 3  Stateful Patterns with Durable Functions

In this chapter, we will discover other fundamental patterns you can implement using Azure Durable Functions. Each of these is managed by the platform precisely in the same way you see for the function chaining pattern. The Durable Functions platform uses the underlying storage to manage each orchestrator's current state and drive the orchestration between the activities.

## Fan-out/fan-in

In the fan-out/fan-in pattern, you must execute several activity functions in parallel and wait for all functions to finish (because, for example, you have to aggregate the results received from them in a single value).

Let’s look at an example. Suppose you want to implement a durable function that implements a backup of a disk folder into a blob storage. In this scenario, you retrieve all the files contained in the source folder, and then, for each file, you copy it to the backup folder in the destination storage. You’ll need to run several functions that implement the copy based on the number of files you find in the source folder.

The following image shows you the scenario.

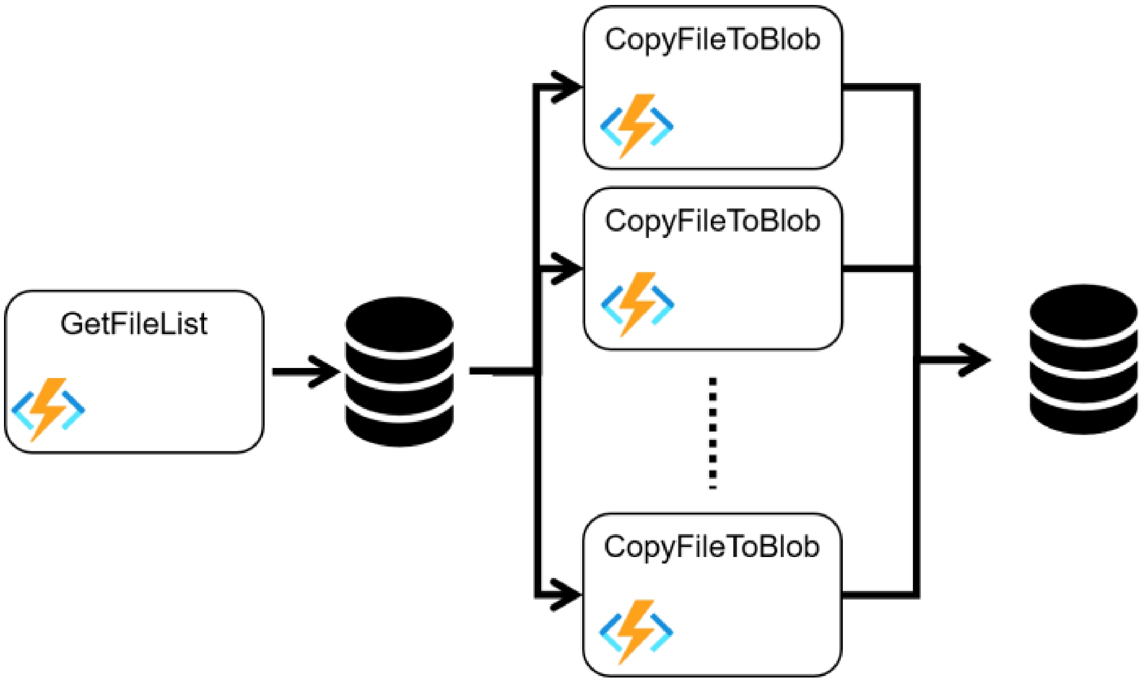


*Figure 22: Fan-out/fan-in*

The backup client starts the orchestrator, and the orchestrator calls the following activities:

- **`GetFileList`**: Retrieves the file list in the source folder and returns it to the orchestrator.
- **`CopyFileToBlog`**: Receives the reference to the file to back up in the destination blob and executes the copy. It will run for each file retrieved in the previous one.

Suppose we try to implement a pattern like this using standard Azure Functions. In that case, we can meet all the difficulties we encountered in the function chaining pattern, and we also have another big obstacle.

Calling multiple activities is relatively simple. For example, you can use a queue and publish one message for each function you want to invoke. But waiting for all the activities executed to finish is too hard. How can you manage the status of all the activities in parallel? You need to know exactly the work progress because when all of these finish their jobs, you need to complete your workflow with the results aggregation.

The Durable Functions platform manages all the work progress of the activity functions for you. It can continue your workflow when all of the activities finish their jobs.

In the following snippet of code, you can see the client function.

*Code Listing 11: Fan-out/fan-in client function*

```
[FunctionName("Backup_Client")]
public async Task<HttpResponseMessage> Client(
    [HttpTrigger(AuthorizationLevel.Function, "post", Route =
"funoutfanin/backup")] HttpRequestMessage req,
    [DurableClient] IDurableOrchestrationClient starter,
    ILogger log)
{
    log.LogInformation($"[CLIENT Backup_Client] --> Backup started!");

    string jsonContent = await req.Content.ReadAsStringAsync();
    try
    {
        var backupRequest =
JsonConvert.DeserializeObject<BackupRequest>(jsonContent);
        if (backupRequest.IsValid())
        {
            var instanceId = await
starter.StartNewAsync("Backup_Orchestrator", backupRequest);

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

            return starter.CreateCheckStatusResponse(req, instanceId);
        }
    }
    catch (Exception ex)
    {
        log.LogError("Error during backup operation", ex);
    }

    return new HttpResponseMessage(System.Net.HttpStatusCode.BadRequest);
}
```

The client manages a **`POST`** request and supposes that it contains the **`BackupRequest`** object based on the JSON schema.

*Code Listing 12: The backup request object schema*

```
{
    "path":"c:\\temp"
}
```

The client starts a new orchestrator, passing the backup request, and the orchestrator looks like the following.

*Code Listing 13: Fan-out/fan-in orchestrator function*

```
[FunctionName("Backup_Orchestrator")]
public async Task<BackupReport> Orchestrator(
    [OrchestrationTrigger] IDurableOrchestrationContext context,
    ILogger log)
{
    log.LogInformation($"[ORCHESTRATOR Backup_Orchestrator] --> id :
{context.InstanceId}");

    var backupRequest = context.GetInput<BackupRequest>();
    var report = new BackupReport();

    var files = await
context.CallActivityAsync<string[]>("Backup_GetFileList",
backupRequest.path);
    report.NumberOfFiles = files.Length;

    var tasks = new Task<long>[files.Length];
    for (int i = 0; i < files.Length; i++)
    {
        tasks[i] =
context.CallActivityAsync<long>("Backup_CopyFileToBlob", files[i]);
    }

    await Task.WhenAll(tasks);

    report.TotalBytes = tasks.Sum(t => t.Result);

    return report;
}
```

The orchestrator executes the following steps:

1. Calls the activity **`Backup_GetFileList`**, passing the path to backup and saving the file list found in the source folder (the activity returns the list).
2. For each file in the file list, the orchestrator creates a **`Task`** object to invoke the activity that copies the file in the destination storage. The orchestrator saves all the tasks in an array.
3. Leverages the **`WhenAll`** method of the **`Task`** class to invoke all the activity. That method finishes when all the tasks in the array finish. This means the orchestrator waits for all the activities to complete.
4. Adds all the results returned by every activity (the copy activity returns the number of bytes copied) to calculate the number of bytes copied during the backup. The **`Result`** property stores the return value of the activity.

You can find the code for the **`GetFileList`** activity in the following snippets of code.

*Code Listing 14: GetFileList activity*

```
[FunctionName("Backup_GetFileList")]
public string[] GetFileList([ActivityTrigger] string rootPath,
    ILogger log)
{
    log.LogInformation($"[ACTIVITY Backup_GetFileList] --> rootPath :
{rootPath}");
    string[] files = Directory.GetFiles(rootPath, "*",
SearchOption.AllDirectories);
    return files;
}
```

The activity receives the disk path from the orchestrator and uses the **`Directory`** class to retrieve the list of files in the folder (in a recursive way).

You can improve the function using a complex object in the signature instead of the simple file path. Using a complex object, you can introduce a set of file filters or a flag that allows you to manage the recursion. We have kept the sample as easy as possible for demo purposes.

The following snippet of code shows the **`CopyFileToBlob`** activity.

*Code Listing 15: CopyFileToBlob activity*

```
[FunctionName("Backup_CopyFileToBlob")]
[StorageAccount("StorageAccount")]
public async Task<long> CopyFileToBlob([ActivityTrigger] string filePath,
    IBinder binder,
    ILogger log)
{
    log.LogInformation($"[ACTIVITY Backup_CopyFileToBlob] --> filePath :
{filePath}");
    long byteCount = new FileInfo(filePath).Length;

    string blobPath = filePath
        .Substring(Path.GetPathRoot(filePath).Length)
        .Replace('\\', '/');
    string outputLocation = $"backups/{blobPath}";

    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;
}
```

The activity receives the file path from the orchestrator. Then, it uses the **`FileInfo`** class to access the file content and copy it (using the dynamic binding provided by the **`IBinder`** interface) to the destination.

The **`IBinder`** interface is an example of dynamic binding in Azure Functions (it is also called imperative approach). This technique allows you to create the binding connection during the runtime phase instead of having the binding declared in the attributes. Thus, you can use it if you need to change the destination service at runtime. Using this approach, you can, for example, choose to connect to a storage blog or to a storage queue every time the function is executed.

The **`StorageAccount`** attribute decorates the activity. It tells the platform that the function uses the storage account defined in the connection string called **`StorageAccount`** stored in the configuration file. The function also recreates, within the backup storage, the same folder hierarchy present in the source folder.

## Async HTTP APIs

One of the most common scenarios when you have a workflow, especially if your workflow is a long-running operation, is to provide a set of APIs to your users that they can use to retrieve the status of the single workflow instance.

An example might be when you need to notify someone that a workflow finished its job.

In the previous chapter, we looked at how the Durable function manages the state. You could access the underlying persistence layer and read the current status. If you are using a storage account as a persistence layer, you must read the **`TaskHubInstances`** table to understand the current state of an orchestrator instance.

Unfortunately, reading the underlying persistence layer is not a good approach, because you are coupled to the persistence schema. You need to modify your code (and retest it) every time something changes in the Durable Function platform.

Fortunately, the scenario is simpler than you might think, because the Durable Function platform provides you with a set of rest APIs you can use to read the status of your orchestration.

Let’s consider the function chaining example we saw in the previous chapter to understand what kind of APIs Durable Functions gives you.

The following code shows the implementation of the client function for the function chaining pattern.

*Code Listing 16: Function chaining client*

```
[FunctionName("OrderManager_Client")]
public async Task<HttpResponseMessage> Client(
    [HttpTrigger(AuthorizationLevel.Function, "post", Route =
"functionchaining/order")] HttpRequestMessage req,
    [DurableClient] IDurableOrchestrationClient starter,
    ILogger log)
{
    log.LogInformation($"[CLIENT OrderManager_Client] --> Order
received!");
    string jsonContent = await req.Content.ReadAsStringAsync();

    try
    {
        var order = JsonConvert.DeserializeObject<Order>(jsonContent);
        var instanceId = await
starter.StartNewAsync("OrderManager_Orchestrator", order);

        log.LogInformation($"Order received - started orchestration with
ID = '{instanceId}'.");

        return starter.CreateCheckStatusResponse(req, instanceId);
    }
    catch (Exception ex)
    {
        log.LogError("Error during order received operation", ex);
    }

    return new HttpResponseMessage(System.Net.HttpStatusCode.BadRequest);
}
```

When you make a **POST** call to the endpoint exposed by that client, you receive a response that looks like the following.

*Code Listing 17: Function chaining client response*

```
{
  "id": "6a3f10a5108a416f87f2d4a2e90265b2",
  "statusQueryGetUri":
"http://localhost:7071/runtime/webhooks/durabletask/instances/6a3f10a5108a4
16f87f2d4a2e90265b2?taskHub=ADFSHub&connection=Storage&code=E1PHYaGwuL1HB8U
Pa6AWWipN7FF8uAd1ZrSDF6O0W8l9I4U9ZIw9aQ==",
  "sendEventPostUri":
"http://localhost:7071/runtime/webhooks/durabletask/instances/6a3f10a5108a4
16f87f2d4a2e90265b2/raiseEvent/{eventName}?taskHub=ADFSHub&connection=Stora
ge&code=E1PHYaGwuL1HB8UPa6AWWipN7FF8uAd1ZrSDF6O0W8l9I4U9ZIw9aQ==",
  "terminatePostUri":
"http://localhost:7071/runtime/webhooks/durabletask/instances/6a3f10a5108a4
16f87f2d4a2e90265b2/terminate?reason={text}&taskHub=ADFSHub&connection=Stor
age&code=E1PHYaGwuL1HB8UPa6AWWipN7FF8uAd1ZrSDF6O0W8l9I4U9ZIw9aQ==",
  "purgeHistoryDeleteUri":
"http://localhost:7071/runtime/webhooks/durabletask/instances/6a3f10a5108a4
16f87f2d4a2e90265b2?taskHub=ADFSHub&connection=Storage&code=E1PHYaGwuL1HB8U
Pa6AWWipN7FF8uAd1ZrSDF6O0W8l9I4U9ZIw9aQ==",
  "restartPostUri":
"http://localhost:7071/runtime/webhooks/durabletask/instances/6a3f10a5108a4
16f87f2d4a2e90265b2/restart?taskHub=ADFSHub&connection=Storage&code=E1PHYaG
wuL1HB8UPa6AWWipN7FF8uAd1ZrSDF6O0W8l9I4U9ZIw9aQ=="
}
```

To generate that kind of response, you can use the method **CreateCheckStatusResponse** of the **IDurableOrchestrationClient** interface. The response provides you with a set of endpoints to get information for the specific orchestrator instance. Before analyzing the behavior of every single endpoint, we take a look at their composition.

As you can see, every single URL provided by the response has the following structure.

```
http://{function FQDN}/runtime/webhooks/durabletask/instances/{InstanceId}?
          taskHub={task hub name}
          &connection=Storage
          &code={system Key}
```

The first part of each URL is the **FQDN** (fully qualified domain name) of our function app. For example, you can see **localhost** in the previous sample because you are debugging your function locally.

The last part of the URL before the query arguments is the instance ID of the orchestrator. If you use a non-existing **instanceId**, you receive an error code 404.

The URL also contains the task hub name we mentioned in the first chapter. Remember that the task hub is where all the information about the history and status of the instances is stored, so the platform needs to know where it can find the instance you want.

The property **`statusQueryGetUri`** is the URL you can use to retrieve the current status of the orchestrator. The query is a **`GET`** request, and the response looks like the following.

*Code Listing 18: The status query response*

```
{
  "name": "OrderManager_Orchestrator",
  "instanceId": "6a3f10a5108a416f87f2d4a2e90265b2",
  "runtimeStatus": "Completed",
  "input": {
    "custName": "……",
    "custAddress": "……",
    "custEmail": "……",
    "cartId": "……",
    "date": "……",
    "price": ……,
    "fileName": null
  },
  "customStatus": null,
  "output": {
    "order": {
      "PartitionKey": "……",
      "RowKey": "……",
      "orderId": "……",
      "custName": "……",
      "custAddress": "……",
      "custEmail": "……",
      "cartId": "……",
      "date": "……",
      "price": ……,
      "isReported": false,
      "fileName": null
    },
    "fullPath": "invoices/5095b268-1a32-43a3-9938-fecce6fdbcd5.txt"
  },
  "createdTime": "2021-06-22T07:00:16Z",
  "lastUpdatedTime": "2021-06-22T07:00:34Z"
}
```

The previous response contains:

- The name of orchestrator (property **`name`**).
- The current status of the orchestrator instance (property **`runtimeStatus`**).
- The input object (property **`input`**). In this property, you can find the object, serialized in JSON format, that you send to the orchestrator at the startup using the client.

- The output object (property **`output`**). You can find the output object in this property, serialized in JSON format. It is the object that the orchestrator returns at the end. This property must be empty if the orchestrator isn't finished yet, or if the orchestrator doesn't have a return object.
- The created time and the last update time of the orchestrator status (properties **`createdTime`** and **`lastUpdatedTime`**).

A particular mention should be reserved for the output object. You need to remember that the orchestrator runs asynchronously compared to the client, and this means the client will complete its execution while the orchestrator has not yet completed its own. The value returned by the orchestrator, therefore, cannot be returned in the client's response. The durable function mechanism is always asynchronous.

You will notice that you receive from the platform all the information you have in the **`TaskHubInstances`** table, so you don't need to directly access that table.

The platform also gives you the ability to make a query over all the orchestrator instances you have in your solution. You can make a **`GET`** request using the following URL.

```
http://{function FQDN}/runtime/webhooks/durabletask/instances?
          taskHub={task hub name}
          &connection=Storage
          &code={system Key}
          &runtimeStatus={status1,status2,...}
          &showInput=[true|false]
          &showHistory=[true|false]
          &showHistoryOutput=[true|false]
          &createdTimeFrom={timestamp}
          &createdTimeTo={timestamp}
          &top={integer}
```

In this kind of query, you don't define the **`orchestratorId`** you are looking for, but you can define some filters:

- **`runtimeStatus`**: You can set one or more runtime statuses (separated by the ",") for the instances to filter the instances in your query. The allowed values are **`Pending`**, **`Running`**, **`Completed`**, **`ContinuedAsNew`**, **`Failed`**, and **`Terminated`**.
- **`showInput`**: If you set this to true, you have the input object in the response. The default is true.
- **`showHistory`**: If you set this to true, the execution history is included in the response. The default is false.
- **`showHistoryOutput`**: If you set this to true, the function outputs are included in the orchestration execution history in the response. The default is false.
- **`createdTimeFrom`**: If you specify this parameter, you filter the orchestrator instances, and you have in the return list only the instances created at or before the value you set. The value must be ISO 8601 timestamp.
- **`createdTimeTo`**: It is similar to the previous, but you filter all the instances created at or before the value you set.
- **`top`**: You can set the number of instances returned by the query. For example, if you set the value to 10, you will receive only the first 10 instances.

Later in this chapter, we will see that you can query your orchestrator instances using a few additional ways.

The **terminatePostUri** property of the function chaining client response gives you the URL to terminate a running orchestrator instance. It is a **POST** request, and if you want, you can set the reason for terminating the orchestration instance. You receive a different HTTP code as response, depending on the current state of the orchestrator. If the orchestrator is running (the status isn't "completed" yet), you receive an HTTP 202 code, while you receive HTTP 404 code if the orchestrator doesn't exist, or HTTP 410 if the orchestrator has completed or failed.

The **purgeHistoryDeleteUri** URL allows you to remove the history and all the related data for a specific orchestrator instance. It is an HTTP **DELETE** request, and its format is the following.

```
http://{function FQDN}/runtime/webhooks/durabletask/instances/{InstanceId}?
          taskHub={task hub name}
          &connection=Storage
          &code={system Key}
```

The **InstanceId** query field is the ID of the instance you want to remove completely.

The platform exposes another URL you can use to remove orchestrator instances. The URL has the following schema.

```
http://{function FQDN}/runtime/webhooks/durabletask/instances?
          taskHub={task hub name}
          &connection=Storage
          &code={system Key}
          &runtimeStatus={status1,status2,...}
          &createdTimeFrom={timestamp}
          &createdTimeTo={timestamp}
```

It is an HTTP **DELETE** request, too, and the query fields you can set are similar to the fields you can use in the query request shown earlier in this chapter.

For this feature, you can receive only two HTTP codes as a response: HTTP 200 if the instance (or instances) is deleted correctly, or HTTP 404 if the instance (or instances) doesn't exist.

The **restartPostUri** URL allows you to restart a specific orchestration instance. It is an HTTP **POST** request, and its format is the following.

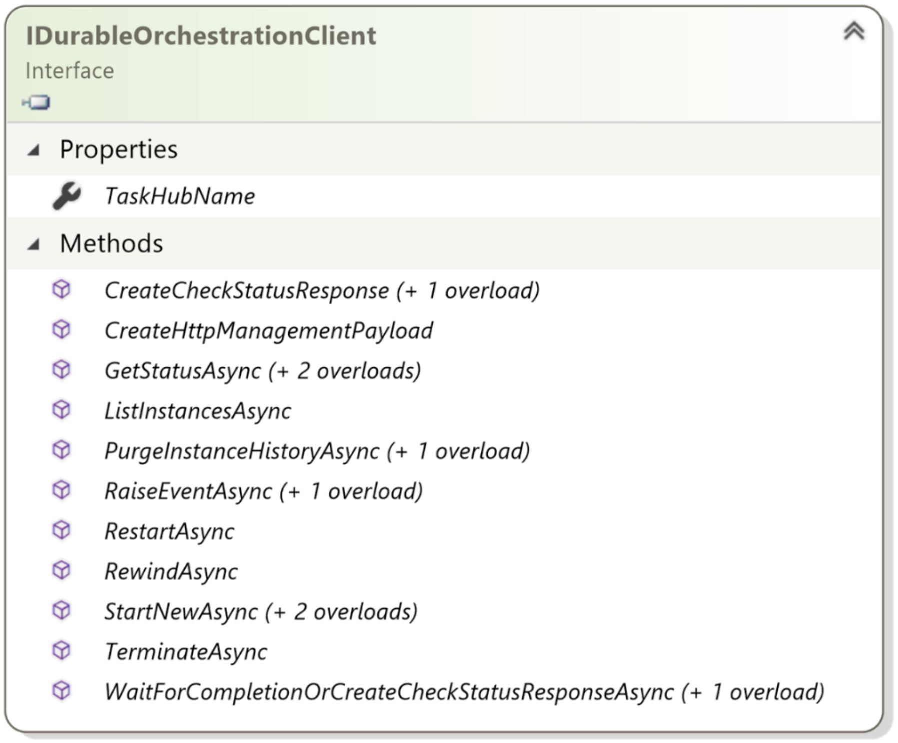
```
http://{function
          FQDN}/runtime/webhooks/durabletask/instances/{InstanceId}/restart?
          taskHub={task hub name}
          &connection=Storage
          &code={system Key}
```

You can use the reason query field to trace the reason for rewinding the orchestration instance. If the restart request is accepted, you receive an HTTP 202, while you receive an HTTP 404 if the instance doesn't exist, or HTTP 410 if the instance is completed or terminated.

The last URL provided by the platform allows you to send an event to an orchestrator. We will see how to use the event approach in the human interaction pattern later in this chapter.

The URLs returned by the **`CreateCheckStatusResponse`** method allow you to interact with your orchestrator instances. However, it isn't a good idea to provide those URLs to your users. You will probably develop a software layer between your user and the orchestrator instances to implement security and access control. You can interact with your orchestrator instances using the **`IDurableOrchestrationClient`** interface in your client function or Azure Function Core Tools.

We used the **`IDurableOrchestrationClient`** interface to start a new orchestration in the previous samples, but it exposes several methods to control our orchestration instances. In the following image, you can see the **`IDurableOrchestrationClient`** interface structure.


IDurableOrchestrationClient
Interface

Properties
TaskHubName

Methods
CreateCheckStatusResponse (+ 1 overload)
CreateHttpManagementPayload
GetStatusAsync (+ 2 overloads)
ListInstancesAsync
PurgeInstanceHistoryAsync (+ 1 overload)
RaiseEventAsync (+ 1 overload)
RestartAsync
RewindAsync
StartNewAsync (+ 2 overloads)
TerminateAsync
WaitForCompletionOrCreateCheckStatusResponseAsync (+ 1 overload)

*Figure 23: The IDurableOrchestrationClient interface structure*

The most important methods are:

- **`ListInstancesAsync`**: This method allows you to execute a query against your orchestration instances like you can using the **`statusQueryGetUri`** URL.
- **`PurgeInstanceHistoryAsync`**: You can use this method to remove the history of and all the data related to an instance.
- **`RaiseEventAsync`**: You can use this method to send an event to an orchestrator instance (we will see this feature in the human interaction pattern).
- **`RestartAsync`**: This method allows you to restart an orchestration.
- **`RewindAsync`**: This one is marked as obsolete: you must use **`RestartAsync`**.
- **`TerminateAsync`**: This method allows you to force the instance termination.

The `WaitForCompletionOrCreateCheckStatusResponseAsync` is a particular method: it waits for an amount of time you can define in one of the arguments (the default is 10 seconds) and returns a client response. If the orchestrator finishes its job during the timeout, the response contains the orchestrator output. Otherwise, it contains the same response you have using the `CreateCheckStatusResponseAsync` method.

Finally, you can manage your orchestration instances using Azure Functions Core Tools. You can retrieve all the options you can execute in the Azure Function Core Tools to manage the orchestration instances using the following command.

```
func durable
```

For example, if you want to retrieve the runtime status for an orchestration, you can run the command:

```
func durable get-runtime-status --id 20ae092dfeb444a993f23b3d511f4532
```

where `id` is the orchestration instance ID.

***Note: You can find more information here about how to use the Azure Function Core Tools to manage your orchestration instances.***

## Monitoring

When we talk about the monitor pattern, we refer to a recurring process in a workflow in which we monitor an external service, periodically clean resources, or something similar.

You can use a standard Azure function with a timer trigger to implement a monitor pattern, but you have some limitations and difficulties using this approach. First of all, as we already saw in the previous patterns, it is not easy to orchestrate different activities invoked by the timer trigger function.

Furthermore, using a timer trigger, you cannot have flexibility on the polling interval. The interval time in a time trigger is static, and you cannot change it without redeploying your function. Using a Durable function, you can easily have different polling intervals for different instances of the same orchestrator.

An example of a monitoring pattern is exposing async HTTP APIs in a software layer and a client that periodically check the status using those APIs.

In this section, we want to analyze the monitor pattern implemented in a Durable function. Let's consider a process that monitors a city's weather conditions and sends an SMS when the weather conditions are identical to those required.

The following figure shows the pattern.

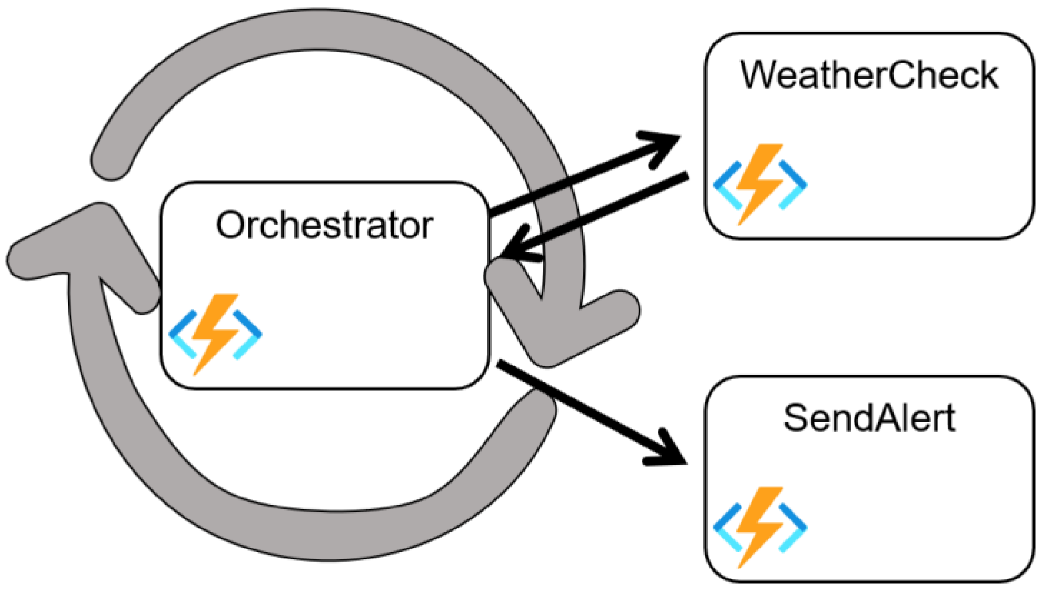


*Figure 24: Monitoring pattern*

As we did for the previous patterns, we need a client function to start the orchestration.

*Code Listing 19: The client for monitoring pattern*

```
[FunctionName("Monitoring_Client")]
public async Task<HttpResponseMessage> Client(
    [HttpTrigger(AuthorizationLevel.Function, "post", Route =
"monitoring/monitor")] HttpRequestMessage req,
    [DurableClient] IDurableOrchestrationClient starter,
    ILogger log)
{
    log.LogInformation($"[CLIENT Monitoring_Client] --> Monitor started!");
    string jsonContent = await req.Content.ReadAsStringAsync();

    try
    {
        var monitorRequest =
JsonConvert.DeserializeObject<MonitorRequest>(jsonContent);
        var instanceId = await
starter.StartNewAsync("Monitoring_Orchestrator", monitorRequest);

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

        return starter.CreateCheckStatusResponse(req, instanceId);
    }
    catch (Exception ex)
    {
        log.LogError("Error during starting monitor orchestrator", ex);
    }

    return new HttpResponseMessage(System.Net.HttpStatusCode.BadRequest);
}
```

The client starts when it receives an HTTP **`POST`** request and tries to retrieve the **`MonitorRequest`** from the request body. The **`MonitorRequest`** has the following schema.

*Code Listing 20: The MonitorRequest JSON*

```
{
  "city": "Rome,IT",
  "durationInMinutes": 60,
  "pollingInMinutes": 5,
  "weatherConditionCheck": "clear sky",
  "phoneNumber": "+39XXXXXXXXXX"
}
```

Using the **`MonitorRequest`** object, you can set the configuration for the specific orchestration.

The properties you can use are:

- **`city`**: The city you want to monitor (for example, **`Rome, IT`**, or **`New York, US-NY`**).
- **`durationInMinutes`**: The duration (in minutes) of the whole monitoring.
- **`pollingInMinutes`**: The time interval (in minutes) between two consecutive monitors.
- **`weatherConditionCheck`**: The desired weather condition (such as **`clear sky`** or **`light rain`**).
- **`phoneNumber`**: The phone number to use for the notification SMS.

So, for example, using the JSON shown in Code Listing 20, you can start a 60-minute monitoring workflow that checks the weather condition of Rome every five minutes and sends an SMS to the phone number set in the JSON when the weather condition becomes “clear sky.”

The orchestrator code is the following.

*Code Listing 21: The monitor pattern orchestrator*

```
[FunctionName("Monitoring_Orchestrator")]
public  async Task Orchestrator([OrchestrationTrigger]
IDurableOrchestrationContext context,
    ILogger log)
{
    log.LogInformation($"[ORCHESTRATOR Monitoring_Orchestrator] --> id :
{context.InstanceId}");

    var request = context.GetInput<MonitorRequest>();

    DateTime endTime =
context.CurrentUtcDateTime.AddMinutes(request.durationInMinutes);

```

```
    while (context.CurrentUtcDateTime < endTime)
    {
        bool isCondition = await
context.CallActivityAsync<bool>("Monitoring_WeatherCheck", request);

        if (isCondition)
        {
            var notificationData = new NotificationData()
            {
                FromPhoneNumber =
this.configuration.GetValue<string>("TwilioFromNumber"),
                PhoneNumber = request.phoneNumber,
                SmsMessage = $"Notification of weather
{request.weatherConditionCheck} for city {request.city}"
            };
            await context.CallActivityAsync("Monitoring_SendAlert",
notificationData);
            break;
        }
        else
        {
            var nextCheckpoint =
context.CurrentUtcDateTime.AddMinutes(request.pollingInMinutes);
            await context.CreateTimer(nextCheckpoint,
CancellationToken.None);
        }
    }
}
```

The orchestrator retrieves the **`MonitorRequest`** object passed by the client and uses the **`duration`** property to calculate the end time for the monitoring. You can see that the orchestrator uses the orchestration context to calculate the end time of the monitoring—the method **`context.CurrentUtcDateTime.AddMinutes()`**.

The orchestrator function must be deterministic: whenever an activity finishes its work, the orchestrator runs again from the beginning, and the state is rebuilt (using the event sourcing approach we saw in the second chapter).

If you use the standard .NET classes to calculate the end time for the monitor workflow, these classes calculate a different date each time the same orchestrator instance runs and can cause problems in reconstructing the state.

For this reason, every time you must do something that can be different every time the orchestrator runs (like a time calculation), you need to use the context or call an activity.

The main part of the orchestrator is the monitoring loop. Inside this loop, the orchestrator calls the **`WeatherCheck`** activity, tests its return value, and if needed, calls the **`SendAlert`** activity and closes the monitoring workflow.

The **WeatherCheck** activity code is shown in the following snippet.

*Code Listing 22: The weather check activity*

```
[FunctionName("Monitoring_WeatherCheck")]
public async Task<bool> WeatherCheck([ActivityTrigger] MonitorRequest
request,
    ILogger log)
{
    log.LogInformation($"[ACTIVITY Monitoring_WeatherCheck] --> request :
{request}");
    try
    {
        var cityCondition = await
this.weatherService.GetCityInfoAsync(request.city);
        if (cityCondition != null)
            return
cityCondition.HasCondition(request.weatherConditionCheck);
    }
    catch (Exception ex)
    {
        log.LogError(ex, $"Error during calling weather service");
    }
    return false;
}
```

The interesting thing in the previous snippet of code is that the activity uses a class (contained in the **weatherService** field) to interact with the external service that gives it the weather condition. It is an alternative example of how you can interact with an external service. You can create your custom binding to interact with an external service, but you can also leverage the dependency injection approach to inject the actual classes into a function.

If you look at the class that contains all the functions related to the monitoring pattern, you can find its constructor.

*Code Listing 23: The function class constructor*

```
private readonly IWeatherService weatherService;
private readonly IConfiguration configuration;

public MonitoringFunctions(IWeatherService weatherService, IConfiguration
configuration)
{
    this.configuration = configuration ?? throw new
ArgumentNullException(nameof(configuration));
    this.weatherService = weatherService ?? throw new
ArgumentNullException(nameof(weatherService));

    this.configuration = configuration;
    this.weatherService.ApiKey = 
configuration.GetValue<string>("WeatherServiceAPI");
}
```

Azure Functions can use the dependency injection provided by the ASP.NET Core platform. You can define a startup class in your Azure Functions project like the following.

*Code Listing 24: The function class construtor*

```
[assembly: WebJobsStartup(typeof(Startup))]

public class Startup : IWebJobsStartup
{
    public void Configure(IWebJobsBuilder builder)
    {
        builder.Services.AddTransient<IWeatherService, 
OpenWeatherMapService>();

        builder.Services.BuildServiceProvider();
    }
}
```

The assembly attribute tells the platform to run the **`Startup`** class when the functions host starts.

In the **`Configure`** method, we can define the resolution for all our classes. For example, in the previous snippet of code, we use the **`AddTransient`** method to configure the dependency resolver to resolve the **`IWeatherService`** interface with the **`OpenWeatherMapService`** class every time a constructor has a dependency with that interface.

***Note: You can find more information here about how you can use dependency injection in Azure Functions.***

The following snippet of code shows the **`SendAlert`** activity.

*Code Listing 25: The SendAlert activity*

```
[FunctionName("Monitoring_SendAlert")]
[return: TwilioSms(AccountSidSetting = "TwilioAccountSid", AuthTokenSetting 
= "TwilioAuthToken")]
public CreateMessageOptions SendAlert([ActivityTrigger] NotificationData 
notificationData,
    ILogger log)
{
    log.LogInformation($"[ACTIVITY Monitoring_SendAlert] --> 
notificationData : {notificationData}");
```

```
    var message = new CreateMessageOptions(new
PhoneNumber(notificationData.PhoneNumber))
    {
        From = new PhoneNumber(notificationData.FromPhoneNumber),
        Body = notificationData.SmsMessage
    };

    return message;
}
```

This activity uses the Twilio binding to send SMSs. It is a return binding, which means you create the message you want to send and return it from the function. After the function is completed without error, the binding implementation sends the message using the account configured in the attribute.

## Human interaction

In some common scenarios, you have the main workflow that an external interaction can influence.

For example, you can imagine an approval process:

- Someone requests approval for something (such as a holiday vacation).
- The approval request waits for an amount of time before it is automatically approved.
- During this period, someone can approve or reject the vacation request.

In that scenario, you need to interact with your workflow from the external. You need to send one or more events to the workflow to change its status (for example, the vacation rejection).

In the following figure, you can see the workflow mentioned before.

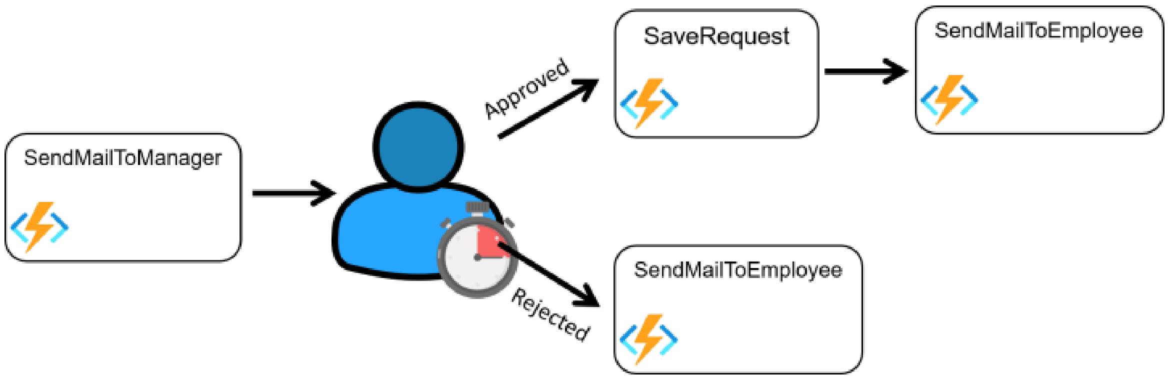


*Figure 25: Human interaction pattern*

What potential issues might we have when implementing this pattern using a standard Azure function?

The first problem appears when you try to implement your function that waits for a while before automatically approving the request. Unfortunately, a standard Azure function cannot run for a long time at will, so we can't start a function and wait.

The second problem you may have if you implement human interaction with an Azure function is how to interact with it if, for example, the manager approves the vacation request.

When an Azure function is running, it is not easy to communicate with it. You could, for example, use a queue and implement your function to check that queue periodically, but your code becomes complex and challenging to maintain.

Using the Durable Functions platform, you can leverage the durable context (through the **`IDurableOchestrationContext`**) that gives you several methods to create timers and interact with a running orchestrator.

We need a client to start the workflow: the employee will request his vacation using the REST API exposed by the following client function.

*Code Listing 26: The human interaction pattern client*

```
[FunctionName("HumanInteraction_Client")]
public async Task<HttpResponseMessage> Client(
    [HttpTrigger(AuthorizationLevel.Function, "post", Route =
"humaninteraction/vacationrequest")] HttpRequestMessage req,
    [DurableClient] IDurableOrchestrationClient starter,
    ILogger log)
{
    log.LogInformation($"[CLIENT HumanInteraction_Client] --> Vacation
requested!");
    string jsonContent = await req.Content.ReadAsStringAsync();

    try
    {
        var vacationRequest =
JsonConvert.DeserializeObject<VacationRequest>(jsonContent);
        if (vacationRequest.IsValid())
        {
            var instanceId = await
starter.StartNewAsync("HumanInteraction_Orchestrator", vacationRequest);
            log.LogInformation($"Monitor started - started orchestration
with ID = '{instanceId}'.");
            return starter.CreateCheckStatusResponse(req, instanceId);
        }
    }
    catch (Exception ex)
    {
        log.LogError("Error during requesting vacation", ex);
    }

    return new HttpResponseMessage(System.Net.HttpStatusCode.BadRequest);
}
```

The client is similar to the other clients you saw previously in this chapter. It is started by an HTTP **`POST`** request and retrieves the vacation request from the body.

The JSON of the vacation request looks like the following.

*Code Listing 27: The vacation request JSON*

```json
{
  "employeeID": "11111",
  "employeeFirstName": "Massimo",
  "employeeLastName": "Bonanni",
  "employeeEmail": "*****@****.***",
  "managerEmail": "*****@****.***",
  "dateFrom": "12/24/2021",
  "dateTo": "12/26/2021",
  "notes": "I need a vacation!!"
}
```

The request contains:

- **`employeeID`**: The identification of the employee.
- **`employeeFirstName`**, **`employeeLastName`**: The first name and last name of the employee.
- **`employeeEmail`**: The employee email. The workflow uses this email to send an email to the employee to communicate the result of the approval process.
- **`managerEmail`**: The email of the employee's manager. The workflow uses this email to notify the manager when the employee requests a vacation.
- **`dateFrom`**, **`dateTo`**: The range of dates of the vacation.
- **`notes`**: The comments related to the request.

The vacation request is elementary because this is just an example, but you can easily understand that the JSON may be more complex.

The orchestrator started by the client is the following.

*Code Listing 28: The human interaction orchestrator*

```csharp
[FunctionName("HumanInteraction_Orchestrator")]
public async Task Orchestrator([OrchestrationTrigger]
IDurableOrchestrationContext context,
    ILogger log)
{
    log.LogInformation($"[ORCHESTRATOR HumanInteraction_Orchestrator] -->
id : {context.InstanceId}");

    var request = context.GetInput<VacationRequest>();
    var response = new VacationResponse()
    {
        request = request,
        instanceId = context.InstanceId
    };

    await context.CallActivityAsync("HumanInteraction_SendMailToManager",
response);

    using (var timeoutCts = new CancellationTokenSource())
    {
        DateTime expiration = context.CurrentUtcDateTime.AddDays(1);
        Task timeoutTask = context.CreateTimer(expiration,
timeoutCts.Token);

        Task approvedResponseTask =
context.WaitForExternalEvent(RequestEvents.Approved);
        Task rejectedResponseTask =
context.WaitForExternalEvent(RequestEvents.Rejected);

        var taskCompleted = await Task.WhenAny(timeoutTask,
approvedResponseTask, rejectedResponseTask);

        if (taskCompleted == approvedResponseTask || taskCompleted ==
timeoutTask) // request approved
        {
            if (taskCompleted == approvedResponseTask)
                timeoutCts.Cancel();
            response.isApproved = true;
            await context.CallActivityAsync("HumanInteraction_SaveRequest",
response);
        }
        else
        {
            timeoutCts.Cancel();
            response.isApproved = false;
        }

        await
context.CallActivityAsync("HumanInteraction_SendMailToEmployee", response);
    }
}
```

The orchestrator calls the **`SendMailToManager`** activity to send the notification email to the employee's manager.

*Code Listing 29: The send email to manager activity*

```
[FunctionName("HumanInteraction_SendMailToManager")]
```

```
public async Task SendMailToManager([ActivityTrigger] VacationResponse
response,
    [SendGrid(ApiKey = "SendGridApiKey")] IAsyncCollector<SendGridMessage>
messageCollector,
    ILogger log)
{
    log.LogInformation($"[ACTIVITY HumanInteraction_SendMailToManager] -->
response : {response}");
    var message = SendGridHelper.CreateMessageForManager(response);
    await messageCollector.AddAsync(message);
}
```

After sending the email to the employee's manager, the orchestrator creates three tasks to manage the waiting time and the two interactions from outside (approval or rejection by the manager).

In particular:

- It uses the **`CreateTimer`** method of the context class to create a durable timer. A durable timer is a feature provided by the platform that allows you to restart the orchestrator after an interval of time. The platform completely manages it: your orchestrator will stop, and when the timer elapses, the platform will recall it.
- It uses the **`WaitForExternalEvent`** method of the context class to have your orchestrator listen for external events. The orchestrator will run again when an accept or reject event is sent. In the previous code, you have two different tasks, one for each type of event. You can define and manage the number of events you want.

In this case, the orchestrator uses the method **`WhenAny`** of the **`Task`** class to continue in the workflow when one of the tasks (the timer or one of the event listeners) finishes its job. The timer task will finish its job when the time is over, while the event listener tasks finish their jobs when one of the events is sent to the orchestrator.

Exiting from the **`WhenAny`** method, the variable **`taskCompleted`** contains the task that completes its job, and we can check it to proceed with the flow. In our sample, if the task completed is the event listener task for the event "accepted" or the timer, we call the **`SaveRequest`** activity to persist the vacation request in the enterprise repository (in our case, it is a storage table).

In the end, whether the request was accepted or rejected, we send the email to the employee using the **`SendMailToEmployee`** activity to notify him of the result of the process.

Another detail to consider is the way the timer is handled. We previously said that the platform actually manages the timer (it is a durable timer), and therefore, when an event arrives (be it **`accepted`** or **`rejected`**), we must signal to the platform that we don't need the timer, and the platform must cancel it.

When we need to cancel the timer (for example, one of the two expected events has arrived), we must execute the **`Cancel`** method.

With Durable Functions, an event is a sort of notification coming from outside your orchestrator composed of a string that defines its name and a possible payload that contains the event data.

You have two different events, called **`approved`** and **`rejected`** in the previous example, and they don't have data associated.

But how I can send an event to an orchestrator?

You can accomplish that task is several ways. You can use one of the HTTP endpoints provided by the platform to manage the orchestrations we saw in the HTTP APIs paragraph.

In particular, if you look at the JSON in Code Listing 17, you can see the property **`sendEventPostUri`**. This property contains a REST POST endpoint you can use to send an event to a specific orchestrator instance.

The signature of the URL is the following.

```
http://{function
          FQDN}/runtime/webhooks/durabletask/instances/{instanceId}/raiseEven
          t/{eventName}?
          taskHub={task hub name}
          &connection=Storage
          &code={system Key}
```

In the previous URL, you address the orchestrator instance using the **`instanceId`** and the event using the **`eventName`** in the URL signature.

The request is a **`POST`**, and its body contains the event payload (if you need it). You receive a response that contains an HTTP 202 code if the event was accepted for processing, an HTTP 400 if the request contains an invalid JSON, an HTTP 404 if the orchestrator doesn't exist, or an HTTP 410 if the instance is completed and cannot process the event.

You can also use the Azure Function Core Tools to send an event to your orchestrations. The following command shows you how you can send the **`approve`** event to the orchestrator.

```
func durable raise-event --id 0ab8c55a66644d68a3a8b220b12d209c --event-name
approve--event-data @eventdata.json
```

The **`eventdata.json`** file contains the payload of our event.

Finally, the third way you can send an event to your orchestrator is by implementing a client function. Inside that function, you can use the **`RaiseEventAsync`** method of the **`IDurableOrchestrationClient`** interface (see Figure 23 for more info).

The following snippet of code shows you how the client approves the vacation request.

*Code Listing 30: The approve vacation client function*

```
[FunctionName("HumanInteraction_ClientApprove")]
public async Task<HttpResponseMessage> ClientApprove(
    [HttpTrigger(AuthorizationLevel.Function, "put", Route =
"humaninteraction/vacationrequest/{instanceId}/approve")]
HttpRequestMessage req
```

```
    string instanceId, [DurableClient] IDurableOrchestrationClient starter,
    ILogger log)
{
    log.LogInformation($"[CLIENT HumanInteraction_ClientApprove] -->
instanceId {instanceId} approved");
    await starter.RaiseEventAsync(instanceId, RequestEvents.Approved,
null);
    return new HttpResponseMessage(System.Net.HttpStatusCode.OK);
}
```

It is an HTTP triggered function that responds to an HTTP **PUT**.

In this example, we do not need data inside the request body because the URL signature contains the instance ID, and the function only handles the **approve** event.

If necessary, the request's body can contain all the data (a serialized object in JSON format) to support the event. This data can be passed to the orchestrator.

The client sends the event to the orchestrator using the **RaiseEventAsync** method. This method has three arguments: the instance ID, the event name (a string), and the object you want to pass to the orchestrator.

In the same way, we implement the client to send the “rejected” event.

*Code Listing 31: The reject vacation client function*

```
[FunctionName("HumanInteraction_ClientReject")]
public async Task<HttpResponseMessage> ClientReject(
    [HttpTrigger(AuthorizationLevel.Function, "put", Route =
"humaninteraction/vacationrequest/{instanceId}/reject")] HttpRequestMessage
req,
    string instanceId, [DurableClient] IDurableOrchestrationClient starter,
    ILogger log)
{
    log.LogInformation($"[CLIENT HumanInteraction_ClientReject] -->
instanceId {instanceId} rejected");
    await starter.RaiseEventAsync(instanceId, RequestEvents.Rejected,
null);
    return new HttpResponseMessage(System.Net.HttpStatusCode.OK);
}
```

One of the most common questions is: what is the best approach for managing one or more events? Must we use the HTTP APIs provided by the platform or implement our clients?

You use the HTTP APIs when you don't want (or don't need) to verify the event payload while you implement your custom client when you want to verify event payload or add some logic before sending the event to the orchestrator.

Implementing a custom client allows you to have more control over the complete end-to-end process.

In the following snippet of code, you can find the implementation of the activities used by the orchestrator to send emails to the employee and the manager and save the request in the database.

*Code Listing 32: The activities of the human interaction pattern*

```
[FunctionName("HumanInteraction_SendMailToManager")]
public async Task SendMailToManager([ActivityTrigger] VacationResponse
response,
    [SendGrid(ApiKey = "SendGridApiKey")] IAsyncCollector<SendGridMessage>
messageCollector,
    ILogger log)
{
    log.LogInformation($"[ACTIVITY HumanInteraction_SendMailToManager] -->
response : {response}");
    var message = SendGridHelper.CreateMessageForManager(response);
    await messageCollector.AddAsync(message);
}

[FunctionName("HumanInteraction_SendMailToEmployee")]
public async Task SendMailToEmployee([ActivityTrigger] VacationResponse
response,
    [SendGrid(ApiKey = "SendGridApiKey")] IAsyncCollector<SendGridMessage>
messageCollector,
    ILogger log)
{
    log.LogInformation($"[ACTIVITY HumanInteraction_SendMailToEmployee] -->
response : {response}");
    var message = SendGridHelper.CreateMessageForEmployee(response);
    await messageCollector.AddAsync(message);

}

[FunctionName("HumanInteraction_SaveRequest")]
public async Task<bool> SaveRequest([ActivityTrigger] VacationResponse
response,
    [Table("vacationsTable", Connection = "StorageAccount")]
IAsyncCollector<VacationResponseRow> vacationsTable,
    ILogger log)
{
    log.LogInformation($"[ACTIVITY HumanInteraction_SaveRequest] -->
response : {response}");

    var vacationRow = new VacationResponseRow(response);
    await vacationsTable.AddAsync(vacationRow);
    return true;
}
```

# Chapter 4  Durable Entities

One of the most important stateful patterns in computer science is the aggregator pattern. In the aggregator pattern, you want to implement something that receives multiple calls from the outside, aggregates all the data it receives in an internal status, and manages the status persistence.

If you want to implement this pattern using Azure Functions, you will run into some issues:

- Every function execution is different from the others. Every function execution is a different instance, potentially running in a different server. For this reason, you need to store its status outside your function, and you need to take care of the persistence layer.
- In this kind of scenario, the data order you receive is important, and you may receive lots of data in a small amount of time. Therefore, you need to guarantee the order of what you receive. Because you cannot avoid multiple instances of the same function related to the same aggregator running simultaneously, you need to manage the concurrency in the saving operation of the state. You need to manage this consistency at the persistence layer.

Imagine you want to implement an Azure function that allows you to store the telemetry data from an IoT field device. The following figure shows the pattern.

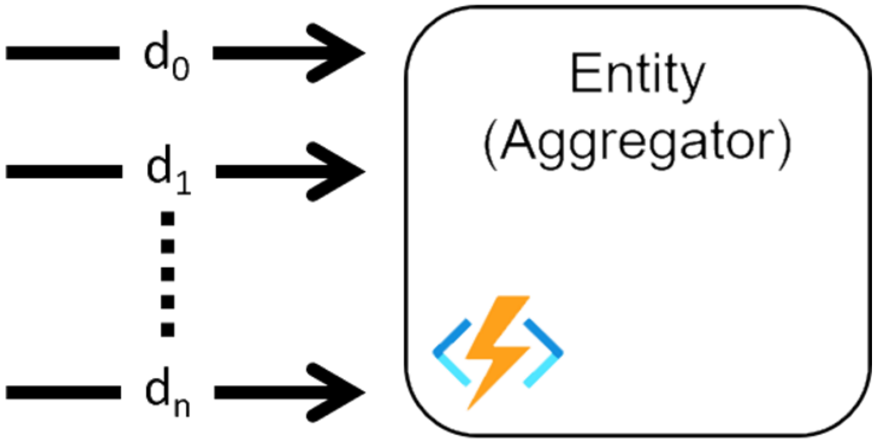


*Figure 26: The aggregator pattern*

We suppose that every device must have these features:

- It receives several telemetries at regular intervals.
- It must store and manage the telemetries in chronological order.
- It can have an internal logic to analyze the telemetry in search of anomalies and, if necessary, send notification of the situation to external services.
- It must expose APIs for retrieving telemetry and for setting configuration.
- We can have devices with different behavior with the least possible impact on the code.

The Durable Functions platform provides us with special durable functions called durable entities, or entity functions. In the following paragraphs, we will see in detail the characteristics of these particular functions and how they can help us. Later in this chapter, we will see the full implementation for an IoT platform based on the durable entities.

## General concepts

Before starting with the implementation of our devices, it is important to clarify some fundamental concepts regarding durable entities.

A durable entity is similar to an orchestrator. It is an Azure function with a special trigger (the **`EntityTrigger`**). Like an orchestrator, a durable entity manages its state abstracting the persistence layer.

You can consider a durable entity like a little service that communicates using messages. It has a unique identity and an internal state, and it exposes operations. It can interact with other entities, orchestrators, or external services.

We saw in the previous chapters that you can access an orchestrator using its orchestrator ID.

You can access a durable entity using a unique identifier called **`entityID`**. This identifier is essentially composed of two strings:

- Entity Name: The name of the entity. This name must be the name you define for the entity function that implements the entity. For example, if we implement our device with an entity function called **`TemperatureDeviceEntity`**, we will use that string as entity name.
- Entity Key: A string that uniquely identifies a specific entity. You can use a friendly name, a GUID, or whatever string you want.

In our example, suppose we implement one device with an entity function called **`TelemetryDeviceEntity`**, and we have two different devices in our environment called **`livingRoom`** and **`kitchen`**. The platform will uniquely identify the devices with the following entity ID: **`@TemperatureDeviceEntity@livingRoom`** and **`@TemperatureDeviceEntity@kitchen`**.

A durable entity can expose several operations, and you need the following information to call a specific operation:

- EntityId: It is the entity ID of the target entity. As we said previously, an entityId allows you to address one and only one entity.
- Operation Name: It is a string that identifies the operation to call. For example, our device can have **`TelemetryReceived`** or **`SetConfiguration`** operations.
- Operation input: This is the parameter (optional) you need to pass to the operation. For example, the **`TelemetryReceived`** operation of our device can have the telemetry object as input.
- Scheduled time: This is optional and allows you to schedule the operation.

You can call an operation from different sources (a client, another entity, or an orchestrator), and you have several ways to do that. We will cover those ways in the "Accessing the entities" section.

An entity operation can change the entity's inner status or call another entity and return a value.

## Entity definition: function-based vs. class-based

You can define your durable entity in two different ways:

- Function-based syntax: You implement your entity as a function, and the operations provided by the entities are dispatched inside that function.
- Class-based syntax: You design your entity as a class with several methods, and every method can be one operation exposed by your entity.

### Function-based syntax

You define a function with a special trigger (**`EntityTrigger`**), and you manage your entity state inside that function.

In the following snippet of code, you can see an example of a simple counter implemented using this approach.

*Code Listing 33: The counter entity implemented with the function-based syntax*

```
[FunctionName("CounterFunctionBased")]
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;
    }
}
```

The function uses **`IDurableEntityContext`** to manage the input data from the caller, the data to return (if you need it), and the entity's state.

The context interface allows you to set and get the status, and it abstracts the persistence layer. You can store only one object in the state (the platform serializes and deserializes it for you before and after every persistence operation). If you need a complex state object, you must declare and use a class to store the state instead of using a base type like an integer.

If you look at Code Listing 33, you can see that the function implements a switch to manage every single operation supported by the entity. Every branch of the switch manages a single operation and uses the context to retrieve the input data, retrieve the state, and save the state.

The context interface also gives you other methods to call orchestrators or other entities and provides you with all the information about the **`EntityId`** invoked. Using the context, you can implement the behavior you prefer for your entity.

This approach works well when:

- You need to provide a dynamic set of operations. Because a string in the function code identifies every operation, you can provide a set of names that change depending on the scenario.
- You have a simple state. The status is simple in our example, but if you have a more complex object, or you need multiple objects in your state, your code grows in complexity.
- You have few operations. If the number of your entity operations is three, like in our example, you can manage them easily, but if the number grows, your code becomes more complex.

The limitations of this approach are:

- Its syntax (based on strings for the operation name) is difficult to maintain and doesn't allow you to have a compile-time check.
- The code isn't readable, and its evolution may be difficult.

You need to manage the retrieving and saving state operations using the context interface. It abstracts the actual persistence operations (you don't see the underlying storage layer), but you explicitly call the **`GetStatus`** and **`SetStatus`** methods.

## Class-based syntax

You can use the function-based syntax for your durable entities if your entity has a simple state or exposes few methods.

In the real world, you often have entities with a complex state (composed of more than one complex object), you need to expose a lot of methods, or more simply, you want to take advantage of the compilation features of the language that you are using such as the compile-time syntax check or the IntelliSense.

In those scenarios, you can implement your durable entity using the class-based syntax. A class-based syntax entity is a POCO class (POCO stands for plain old CLR object) with properties, methods, and one entity function (similar to the entity function you see earlier in the function-based syntax).

This class doesn't require a special hierarchy (you can create your own class structure). It only requires that:

- It must be constructible (it must have at least one public constructor).
- It must be JSON serializable.

The platform automatically serializes and deserializes every single entity instance to store the status on the underlying storage layer. All its serializable properties represent the inner state (so you can design the entity state as a composition of complex objects).

You have some limitations on how you can implement the methods, and in particular:

- A method must have zero or one argument.
- You cannot use overloading, so you cannot have two methods with the same name but different types or arguments.
- You cannot use generics in the method definition.
- Arguments and return values must be JSON serializable.

The entity class exposes an entity function you can use to call the methods provided by the class. When you use your entity in a durable context, you don't call its methods directly. You call the entity function, and it invokes your particular method.

You can read or update the entity state by reading or changing the value in one or more properties in an entity method. The platform stores the state in the storage immediately after your method is finished. Moreover, in a method, you can perform I/O operations or computational operations: the limits are the common limits of Azure functions.

Finally, in an entity method, you can use the entity context (for example, to call another entity or create a new orchestration) through the **`Entity.Context`** object.

The following code listing shows you the implementation of the **`Counter`** entity.

*Code Listing 34: The Counter entity implemented with the function-based syntax*

```
[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>();
}
```

As you can see, the device class consists of:

- A set of properties: They map the inner status you want to manage. In the previous code, for example, we define the **`CurrentValue`** property to store the value of the counter. You can use complex objects to store complex values in your state.
- A set of public methods: They can be invoked from outside the entity using the entity function.
- A set of private methods: You can use private methods to implement internal features. We don’t implement any private methods in the previous example because it is simple.
- An entity function: The platform uses it to manage the calls from outside the entity directed to the entity.

If you look at the entity function, you can see that this function uses the durable context to dispatch the calls received by the entity to the correct method without implementing any switch in the code. The **`DispatchAsync`** method uses the call context provided by the platform to try to invoke the method in the class. Here you don't have any string as happened in the function-based syntax, so the possibility for error is less.

The name of the entity function (declared in the **`FunctionName`** attribute) is also the entity's name. You can have different names for the class and the entity.

A durable entity can implement an interface. In that way, you can create a polymorphism in the durable entities platform.

The following code listing shows you the interface for the counter.

*Code Listing 35: The Counter interface*

```
public interface ICounter
{
    void Add(int amount);
    Task AddAsync(int amount);
    Task ResetAsync();
    Task<int> GetAsync();
}

[JsonObject(MemberSerialization.OptIn)]
public class Counter : ICounter
{
    [JsonProperty("value")]
    public int CurrentValue { get; set; }

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

    public Task AddAsync(int amount)
    {
        this.Add(amount);
        return Task.CompletedTask;
    }
    public Task ResetAsync()
    {
        this.CurrentValue = 0;
        return Task.CompletedTask;
    }

    public Task<int> GetAsync() => Task.FromResult(this.CurrentValue);

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

When you use an interface, you have some other limitations on the methods you can define:

- The entity interface can only have methods.
- The entity interface cannot use generic parameters.
- The entity interface methods must not have more than one parameter.
- An interface method must return a **`void`**, **`Task`**, or **`Task<T>`** value.

If you define a **`void`** method, you can only call that method using a fire-and-forget pattern (a one-way call). A **`Task`** or **`Task<T>`** method can call both one-way and two-way patterns. You call those methods using the two-way pattern when you want to receive a result. We look at both patterns in the next section.

## Accessing the entities

You can invoke an entity in two different ways:

- Calling: It is a two-way (round-trip) communication. You call one method exposed by the entity and wait for the method completion. The entity can return a value or not, or it can throw an exception (an error code in JavaScript). In the case of the exception, it will be caught by the caller.
- Signaling: It is a one-way (fire and forget) communication. You call a method exposed by the entity, but you don't wait for a response. In this scenario, the platform delivers your call to the entities, but you cannot know if the method throws an error.

An entity can be accessed by client functions, orchestrator functions, and other entities, and the way to call the entity depends on the caller.

A client function can call an entity using a one-way approach or can retrieve the entity state.

An orchestrator function can use both one-way and two-way patterns.

An entity can only signal another entity.

## Accessing the entity from a client function

A client function can signal an entity or read its status.

The following snippet of code shows you how you can use the one-way approach inside a client function.

*Code Listing 36: A client function signals an entity*

```
[FunctionName("AddFromQueue")]
public static Task AddFromQueue([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 client function is triggered by a message in the **`durable-function-trigger`** queue.

It creates the **`EntityId`** object for the entity to signal to and calls the **`SignalEntityAsync`** method provided by the **`IDurableEntityClient`** interface.

The entity invocation is completely asynchronous: you cannot know when the destination entity will process the request.

The **`IDurableEntityClient`** interface used by the client provides you with all the methods you need to interact with an entity inside a client.

In the following figure, you can see the methods exposed by the interface.

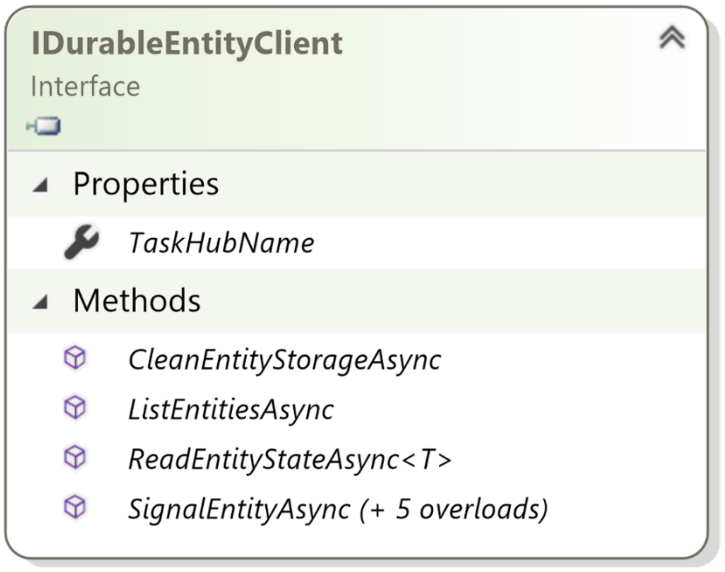


*Figure 27: The IDurableEntityClient interface structure*

It is important to know how this interface works.

The **`SignalEntityAsync`** allows you to create a one-way communication between your client and an entity, and you have six different overloads of this method. Two of those overloads (one is used in the previous snippet of code) allow you to call a method of the entity knowing the **`entityId`** and the method's name (a string). One of those also allows you to signal the entity delayed: you can specify a **`DateTime`** object, and the method will signal the entity at that time.

Both the overloads use a string to identify the method, which can create some issues when you refactor your code. If you change the name of an entity method, the compiler will not give you any information that the name you are using in the **`SignalEntityAsync`** is wrong. You can determine that only at runtime.

If your entity implements an interface (as mentioned earlier in this chapter), you can use one of the other signatures of the method.

*Code Listing 37: A client function signals an entity through an interface*

```
[FunctionName("AddFromQueueWithInterface")]
public static Task AddFromQueueWithInterface([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<ICounter>(entityId, c =>
c.Add(amount));
}
```

In this case, you use a lambda expression to identify the entity method you want to signal, and the compiler will give you a compile-time error if you miss the method name. This is one reason why defining and using an interface for your entities is a best practice.

Using the **`IDurableEntityClient`** interface, you can also read the state of an entity.

*Code Listing 38: A client function reads the entity state*

```
[FunctionName("GetCounterState")]
public static async Task<IActionResult> GetCounterState(
        [HttpTrigger(AuthorizationLevel.Function, Route =
"counters/{counterName}")] HttpRequest req,
        string counterName,
        [DurableClient] IDurableEntityClient client)
{
    var entityId = new EntityId(nameof(Counter), counterName);
    EntityStateResponse<JObject> stateResponse = await
client.ReadEntityStateAsync<JObject>(entityId);
    return new OkObjectResult(stateResponse.EntityState);
}
```

The **`EntityStateResponse`** object contains the state of the entity. You can use your class as a generic parameter in the **`ReadEntityStateAsync`** method if you want to deserialize the status in your custom object. In the sample, we retrieve the status as **`JObject`**. The **`JObject`** class can contain complex objects serialized in JSON format.

If you look at Figure 27, you can see another two methods exposed by the **`IDurableEntityClient`** interface. The **`ListEntitiesAsync`** method allows you to retrieve the list of all the entities in the platform.

*Code Listing 39: A client function to retrieve all the counter entities*

```
public class CounterQueryState
{
    public string Key { get; set; }
    public int Value { get; set; }
}

[FunctionName("CounterList")]
public static async Task<IActionResult> CounterList(
        [HttpTrigger(AuthorizationLevel.Function, Route = "counters")]
HttpRequest req,
        [DurableClient] IDurableEntityClient client)
{
    var responseList = new List<CounterQueryState>();

    var queryDefinition = new EntityQuery()
    {
        PageSize = 100,
        FetchState = true,
        EntityName = nameof(Counter)
    };

    do
    {
        EntityQueryResult queryResult = await 
client.ListEntitiesAsync(queryDefinition, default);

        foreach (var item in queryResult.Entities)
        {
            var counterState = item.State.ToObject<CounterQueryState>();
            counterState.Key = item.EntityId.EntityKey;
            responseList.Add(counterState);
        }

        queryDefinition.ContinuationToken = queryResult.ContinuationToken;
    } while (queryDefinition.ContinuationToken != null );

    return new OkObjectResult(responseList);
}
```

You create an **`EntityQuery`** object that allows you to control how you execute the query against the platform. The **`EntityQuery`** object allows you to define the page size for each fetch operation (the query in the platform is always paginated), the name of the entity you want to search (if you want), if the query results contain the state of the entities, and so on.

Every time you call the **`ListEntitiesAsync`** method, you retrieve a page of the query, and you need to provide the continuation token returned by the previous invocation. You can know if you reach the last page by checking the continuation token. When it is null, you've retrieved the last page of the search.

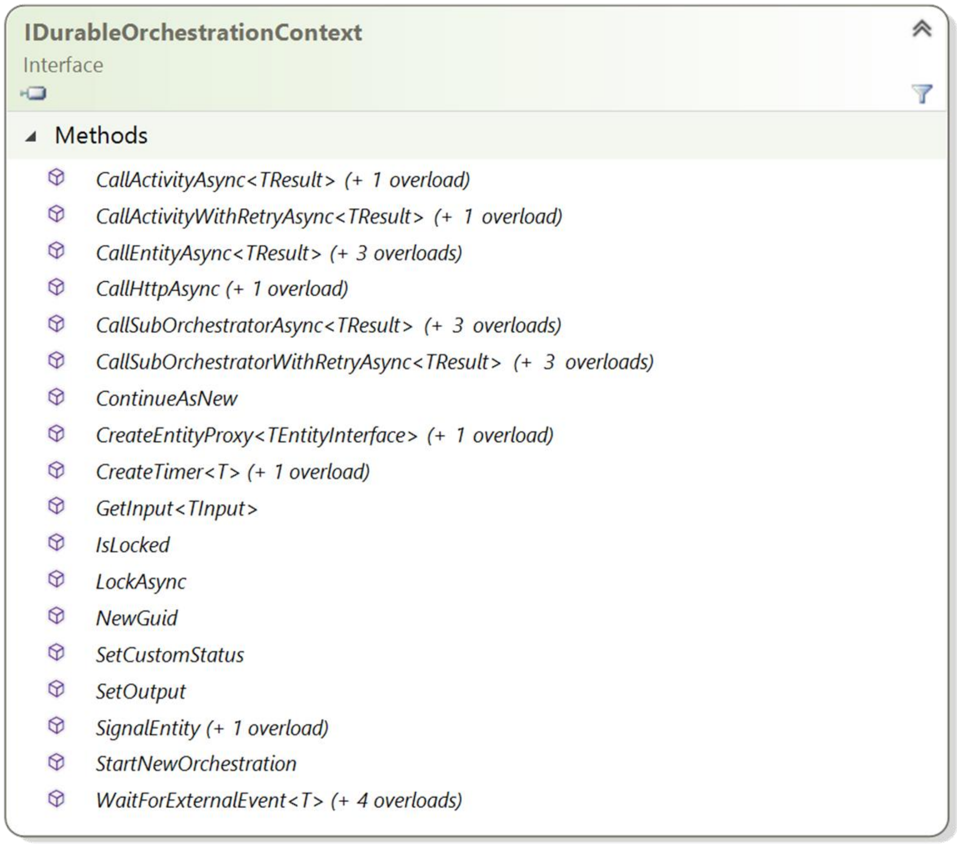
The last method of the **`IDurableEntityClient`** interface is **`CleanEntityStorage`**. This method allows you to remove all the empty entities in your platform. An empty entity is an entity with no state, not used by other entities (with no active locks) and idle for a configurable amount of time in the durable function settings. This method allows you to clear your persistence storage safely.

## Accessing the entity from an orchestrator function

When you use an orchestrator to interact with an entity, you have the most flexibility. You can use both the patterns one-way and two-way inside a workflow. An orchestrator can signal an entity without waiting for the response. Still, because the platform can manage the workflow state and restart the orchestrator, it can wait for a response from the entity. From this point of view, calling an entity is equal to calling an activity.

The orchestration context (provided by the platform through the **`OrchestrationTrigger`**) gives you all the methods you need to interact with entities.

In the following figure, you can see the structure of the **`IDurableOrchestrationContext`** interface.



CallActivityAsync<TResult> (+ 1 overload)
CallActivityWithRetryAsync<TResult> (+ 1 overload)
CallEntityAsync<TResult> (+ 3 overloads)
CallHttpAsync (+ 1 overload)
CallSubOrchestratorAsync<TResult> (+ 3 overloads)
CallSubOrchestratorWithRetryAsync<TResult> (+ 3 overloads)
ContinueAsNew
CreateEntityProxy<TEntityInterface> (+ 1 overload)
CreateTimer<T> (+ 1 overload)
GetInput<TInput>
IsLocked
LockAsync
NewGuid
SetCustomStatus
SetOutput
SignalEntity (+ 1 overload)
StartNewOrchestration
WaitForExternalEvent<T> (+ 4 overloads)

*Figure 28: The IDurableOrchestrationContext interface structure*

In the following snippet of code, you can see an example of an orchestrator that calls an entity waiting for the response, and then interacts with the same entity with a one-way approach.

*Code Listing 40: Interaction between an orchestrator and an entity*

```
[FunctionName("CounterOrchestration")]
public static async Task CounterOrchestration([OrchestrationTrigger]
IDurableOrchestrationContext context)
{
    var counterName = context.GetInput<string>();
    var entityId = new EntityId(nameof(Counter), counterName);

    int currentValue = await context.CallEntityAsync<int>(entityId, "Get");
    if (currentValue < 10)
    {
        context.SignalEntity(entityId, "Add", 1);
    }
}
```

The orchestrator executes these operations:

- Retrieves the name of the counter entity passed by the client (**`GetInput`** method) with which to interact.
- Creates the **`entityId`** to identify the entity with which to interact uniquely.
- Calls the **`Get`** method of the entity and waits for the result (**`CallEntityAsync`** method).
- Based on the result of the previous call, signals the entity for the **`Add`** method (**`SignalEntity`** method).

Code Listing 40 is easy to understand, but it has a big issue. The names of the operations you want to call or signal in the entity are strings, and you cannot leverage the compiler checking at compile time. If you make a mistake entering one of the names, you have an error at runtime.

To avoid this, you can use the entity interface and you can create a proxy for the entity you want to interact with, as shown in the following code.

*Code Listing 41: Interaction between an orchestrator and an entity using the entity interface*

```
[FunctionName("CounterOrchestrationWithProxy")]
public static async Task
CounterOrchestrationWithProxy([OrchestrationTrigger]
IDurableOrchestrationContext context)
{
    var counterName = context.GetInput<string>();
    var entityId = new EntityId(nameof(Counter), counterName);

    var entityProxy = context.CreateEntityProxy<ICounter>(entityId);

    int currentValue = await entityProxy.GetAsync();
    if (currentValue < 10)
    {
        entityProxy.Add(1);
    }
}
```

The **`CreateEntityProxy`** method enables you to create a proxy that exposes the entity interface you want (in the previous snippet of code, the **`ICounter`** interface). You can use this proxy to call the entity's methods directly.

As you can see in the snippet, the **`CallEntityAsync`** method of the orchestrator context is substituted with the **`Get`** method of the **`ICounter`** interface, and the **`SignalEntity`** method with the **`Add`**. If you save the result of the method (like in the **`Get`** method of the previous snippet), you are waiting for the response, so you are using a two-way approach. If you don't store the result of a method (like in the **`Add`** method invocation), you are using the one-way pattern.

The code is clearer and more maintainable, and you avoid having runtime errors by mistaking the names of the methods.

Another helpful scenario may be the coordination between entities. In some use cases, you must interact with more than one entity simultaneously, and you want other functions not to modify the entities involved in your task.

The orchestration context interface provides you with a lock implementation to implement those scenarios. At the time of writing this book, unfortunately, this feature is only present for durable functions written in .NET.

In the following snippet of code, you can see an example of a lock between two counter entities.

*Code Listing 42: Use of the LockAsync method of the IDurableOrchestrationContext*

```
[FunctionName("AddCounterValue")]
public static async Task AddCounterValue([OrchestrationTrigger]
IDurableOrchestrationContext context)
{
    var counters = context.GetInput<string>();
    if (string.IsNullOrWhiteSpace(counters))
        return;
    var counterNames = counters.Split("|");
    if (counterNames.Count() != 2)
        return;

    var sourceEntityId = new EntityId(nameof(Counter), counterNames[0]);
    var destEntityId = new EntityId(nameof(Counter), counterNames[1]);

    using (await context.LockAsync(sourceEntityId, destEntityId))
    {
        ICounter sourceProxy =
context.CreateEntityProxy<ICounter>(sourceEntityId);
        ICounter destProxy =
context.CreateEntityProxy<ICounter>(destEntityId);

        var sourceValue = await sourceProxy.GetAsync();

        await destProxy.AddAsync(sourceValue);
        await sourceProxy.AddAsync(-sourceValue);
    }
}
```

The orchestrator retrieves the value in the state of a source counter entity, adds that value to a destination counter entity, and subtracts it from the source entity. The platform will perform all the operations with a lock on the source and destination entities. This means that other orchestrations or entities cannot interact with the same entities while the lock is active.

The **`LockAsync`** method creates a critical section in the orchestrator and returns an **`IDisposable`** object. You can use the using construct or call the **`Dispose`** method explicitly to end the critical section and release all the locks on the entities involved in the operation. When you create a lock with the **`LockAsync`** method, the platform sends a "lock" message to all the entities and waits for the "lock acquired" confirmation.

When an entity is locked, every other operation it receives will be placed in the pending operation queue. When the platform releases the lock, the entity will process all the pending operations. The platform persists the lock in the entity state.

Remember that this isn't a transaction. The critical section doesn't automatically roll back the state of the entities involved in the section. It just prevents other functions from modifying the state while you are working on those entities. You must implement your rollback logic if you need it.

There are some limitations with critical sections:

- You cannot create a critical section in another critical section or nest critical sections.
- An orchestrator cannot call another orchestrator in a critical section.
- Inside a critical section, you can call or signal only the entities you locked.
- You cannot call the same entities with parallel calls in a critical section.

If you violate one of the previous limitations, you receive a runtime error (a **`LockingRulesViolationException`**).

## Accessing an entity from an entity

You can interact with an entity from another entity, but you can only use the one-way pattern: an entity can only signal another entity.

The entity context (both in a function-based syntax using the trigger of the function, and in the class-based syntax using the **`Entity.Current`** property) provides you with the **`SignalEntity`** method to achieve that.

To give an example, suppose you want to monitor how many times your counter is called by the **`ResetAsync`** method. Of course, you can add a new property in the state that takes care of that number, but this information is related to the counter without belonging to it. It is monitoring information, so the best approach may be to create another entity called **`MonitorEntity`** to take care of all the monitoring information you want to store.

In the following snippet of code, you can see a simple implementation of the **`MonitorEntity`**.

*Code Listing 43: A simple MonitorEntity entity*

```
public interface IMonitorEntity
{
    void MethodCalled(string methodName);
}

[JsonObject(MemberSerialization.OptIn)]
public class MonitorEntity : IMonitorEntity
{
    [JsonProperty("methodsCounters")]
    public Dictionary<string, int> MethodsCounters { get; set; }
    public void MethodCalled(string methodName)
    {
        if (MethodsCounters == null)
            MethodsCounters = new Dictionary<string, int>();

        if (MethodsCounters.ContainsKey(methodName))
            MethodsCounters[methodName]++;
        else
            MethodsCounters[methodName] = 1;
    }

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

Now we can modify the method **`ResetAsync`** of the **`Counter`** entity to signal the **`MethodCalled`** operation of the **`MonitorEntity`** every time the **`ResetAsync`** is executed.

*Code Listing 44: Signaling an entity inside an entity*

```
public Task ResetAsync()
{
    this.CurrentValue = 0;

    // Monitoring all the reset operation signaling in a MonitorEntity.
    // The monitoring entity has the same key of the counter.
    var monitorEntityId = new EntityId("MonitorEntity",
Entity.Current.EntityId.ToString());
    Entity.Current.SignalEntity<IMonitorEntity>(monitorEntityId, m =>
m.MethodCalled("ResetAsync"));

    return Task.CompletedTask;
}
```

## Entity state management

The platform abstracts our entities' state persistence, which means the platform saves and retrieves each entity's status when we address it by calling any method.

As we said earlier in this paragraph, our entity class must be serializable because the durable function runtime uses the Json.NET library to serialize the entire class in the underlying storage.

***Note: The Json.NET package is available [for download here](), and you can read full documentation about it [here]().***

To control how the runtime serializes our class, we can use the attributes present in the Json.NET library. If we look at Code Listing 35, we can see that in the **`Counter`** class, we use two different attributes:

- **`JsonObject`**: This attribute allows us to control how Json.NET will serialize the class in the storage. With the **`MemberSerialization`** equal to **`OptIn`**, only the class members marked with the attribute **`JsonProperty`** (or **`DataMember`**) will be serialized. **`OptOut`** allows you to serialize all the members except for that member marked with the **`JsonIgnore`** attribute.
- **`JsonProperty`**: This attribute allows you to explicitly mark the member you want to serialize (and if you want to define the name of the JSON property in the serialization).

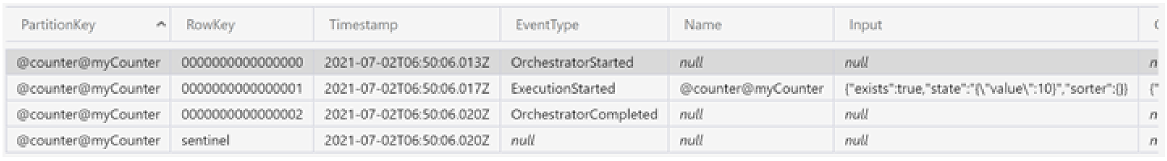
You can use all the serialization attributes provided by Json.NET. For example, you can use the **`JsonIgnore`** attribute to prevent Json.NET from serializing a specific member.

The previous attributes aren't mandatory. Our entity is a standard class, and the Json.NET manages it even if you don't decorate the class member. If you like, you can use the **`DataContract`** and **`DataMember`** attributes, as shown in the following code.

*Code Listing 45: The counter class decorated using DataContract*

```
[DataContract]
public class CounterDataMember
{
    [DataMember(Name = "value")]
    public int CurrentValue { get; set; }

    .....
}
```

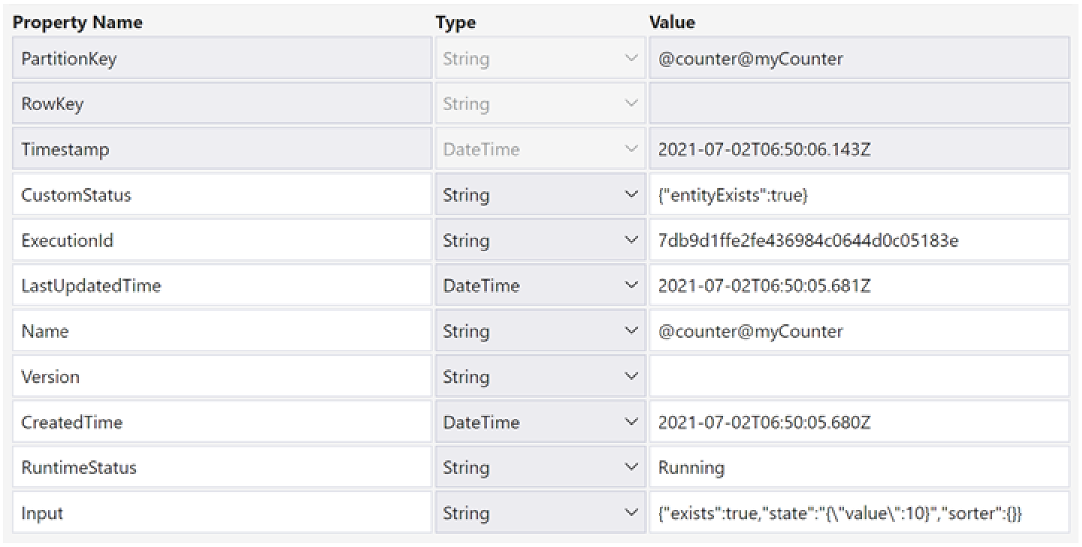
The runtime manages the entity function executions in the same way it manages an orchestrator. It uses the **`TaskHubHistory`** table to store each step the entities did. So, for example, if we call the **`Add`** method of the **`Counter`** entity **`myCounter`**, the result is shown in the following figure.



| PartitionKey | RowKey | Timestamp | EventType | Name | Input | ( |
|---|---|---|---|---|---|---|
| @counter@myCounter | 0000000000000000 | 2021-07-02T06:50:06.013Z | OrchestratorStarted | *null* | *null* | *n* |
| @counter@myCounter | 0000000000000001 | 2021-07-02T06:50:06.017Z | ExecutionStarted | @counter@myCounter | {"exists":true,"state":"{\"value\":10}","sorter":{}} | {" |
| @counter@myCounter | 0000000000000002 | 2021-07-02T06:50:06.020Z | OrchestratorCompleted | *null* | *null* | *n* |
| @counter@myCounter | sentinel | 2021-07-02T06:50:06.020Z | *null* | *null* | *null* | *n* |

*Figure 29: myCounter history table for Add execution*

The runtime uses the **`TaskHubInstances`** table to store the current status of every entity. In the following figure, you can see the **`myCounter`** status after the **`Add`** invocation.



| Property Name | Type | Value |
|---|---|---|
| PartitionKey | String | @counter@myCounter |
| RowKey | String | |
| Timestamp | DateTime | 2021-07-02T06:50:06.143Z |
| CustomStatus | String | {"entityExists":true} |
| ExecutionId | String | 7db9d1ffe2fe436984c0644d0c05183e |
| LastUpdatedTime | DateTime | 2021-07-02T06:50:05.681Z |
| Name | String | @counter@myCounter |
| Version | String | |
| CreatedTime | DateTime | 2021-07-02T06:50:05.680Z |
| RuntimeStatus | String | Running |
| Input | String | {"exists":true,"state":"{\"value\":10}","sorter":{}} |

*Figure 30: The current status of myCounter in the TaskHubInstances*

As you can see, the runtime uses the **`Input`** field to store the serialization of your class (in the state property of the complex object stored in the field).

## State and versioning

In the real world, your project evolves, and you may need to modify the class that implements your entity. As the entity state is serialized in JSON format within the storage, you need to pay attention to some scenarios:

- You add a new property in your class. In this case, that property isn't in the current status stored in the storage, and it will assume the default value when your entity is recovered.
- You remove a property in the class, and the property is in the serialization on the storage. In this case, the value in that property will be lost next time you interact with that entity.
- You rename a property in the class. The result is the same as when removing a property, and the "new" property will assume the default value the first time you interact with it.
- You change the type of property. In this case, you can have two different behaviors. If the runtime can deserialize the old type in the new one (for example, you have an integer property and change it to double), it can recreate the status, and nothing happened. If the runtime cannot deserialize the old type in the new one, an exception will occur when you try to interact with the entity.

Because the runtime is using Json.NET, you can use all the options in Json.NET to control the serialization of the objects. You can write your own code to deserialize the JSON in the status as you prefer.

# Chapter 5  Sample Scenario: Backend for IoT Devices

In this chapter, we want to implement a backend infrastructure based on Durable Functions (and durable entities) to manage the actual state of a set of IoT devices.

The following figure shows you the scenario.

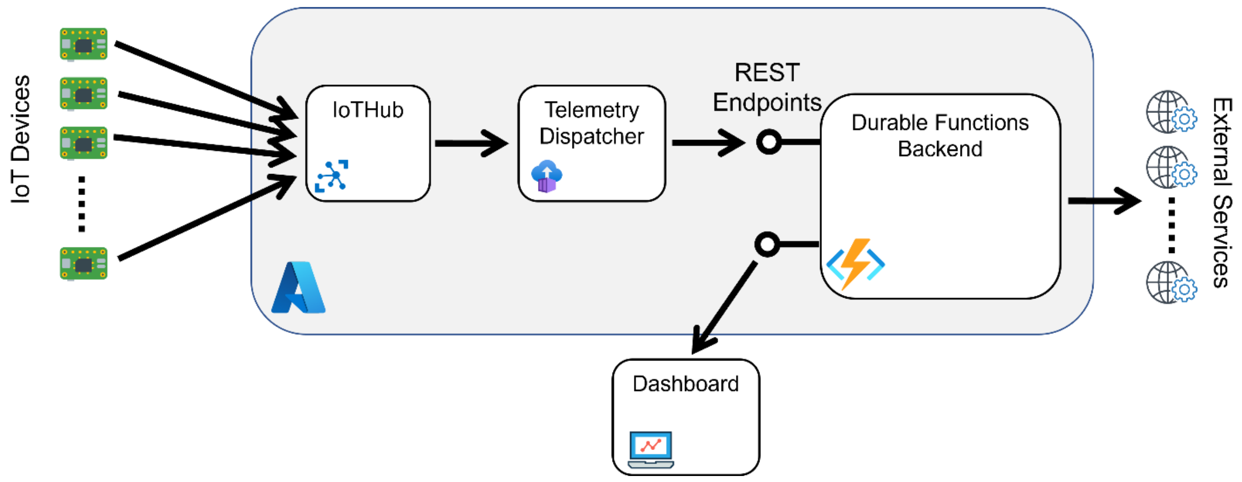


*Figure 31: The architecture schema for the IoT backend platform*

The scenario shown in Figure 31 is a classic IoT scenario. A set of devices produce telemetries (in our example, we suppose that they produce temperature telemetries, but we try to generalize the kind of telemetry). These telemetries must be ingested in our cloud platform to store and use data. We use an IoTHub service to ingest data, and we must implement a dispatcher layer to get telemetries from IoTHub and dispatch them to the correct device in the backend layer.

We don't care about the IoTHub and the dispatcher in this example because they aren't topics related to Durable Functions. You can find several implementations of these components in Microsoft documentation.

Let’s suppose that the backend layer must have these requirements:

- The backend must expose one or more REST endpoints to send telemetries to the devices.
- The backend must expose one or more REST endpoints to retrieve the list of the devices in the platform, the last telemetries received by a single device. A control dashboard can use these endpoints to show the device status.
- The backend must expose one or more REST endpoints to configure every single device in terms of the amount of telemetry to be kept in memory or the thresholds to use to send alarms to external services.
- The type of devices the platform supports can change over time and adding new types of devices must be easy. For example, in the second version, we could add devices that manage telemetry containing humidity values.

- The backend must interact with external services to start workflows (for example, sending an SMS when the temperature of a device goes over a threshold you set in the configuration). This interaction can change over time, and its evolution must be simple.
- The runtime we use must abstract the persistence operations for saving status for the devices.

During the example explanation, we try to emphasize how we implement every single requirement.

We can design our solution with three layers, as the following figure shows.

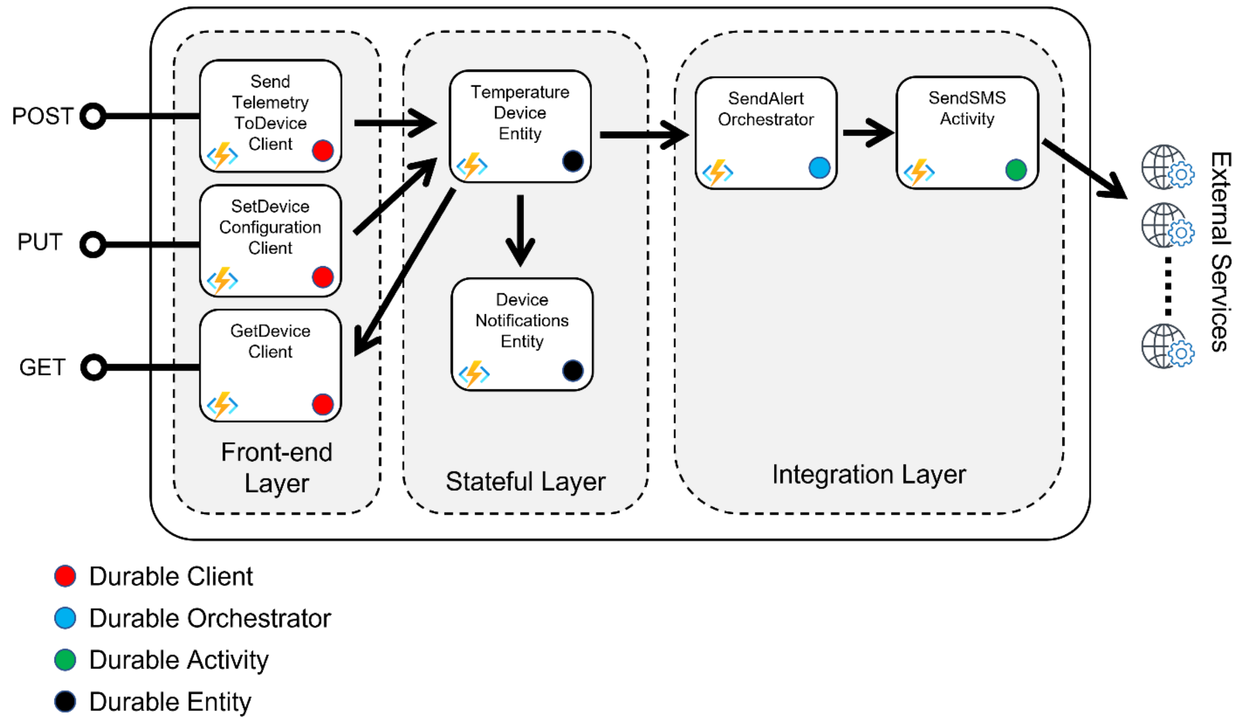


*Figure 32: The layers of the IoT platform*

The front-end layer contains all the REST endpoints to expose all the features requested in the requirements. This layer is composed of durable clients that interact with the stateful layer.

The stateful layer contains all the objects that manage the status of the devices. This layer is composed of durable entities, and each entity manages a device in the field.

Finally, the integration layer contains all the workflows invoked by the stateful layer and manages the integration with the external services. In this layer, you can find orchestrators and activities.

## Front-end layer

The front-end layer contains the client functions that implement the APIs you can use to send telemetry to the devices or retrieve information from them.

These are the clients in the front-end layer:

- **SendTelemetryToDevice**: Allows you to send telemetry to one of the devices in the stateful layer.
- **GetDevices**, **GetDevice**: Allows you to retrieve the list of the devices in the solution or to get information about a specific device.
- **SetConfiguration**: Allows you to configure a specific device, setting the telemetry thresholds or alert details.
- **GetDeviceNotifications**: Allows you to retrieve the list of notifications for a single device.

Each of those client functions uses one of the previous chapters' communication patterns to communicate with the stateful layer composed of entities.

The following snippet of code shows you the implementation of **SendTelemetryToDevice** client functions.

*Code Listing 46: SendTelemetryToDevice function*

```
[FunctionName(nameof(SendTelemetryToDevice))]
public async Task<IActionResult> SendTelemetryToDevice(
    [HttpTrigger(AuthorizationLevel.Function, "post", Route =
"devices/{deviceId}/telemetries")] HttpRequest req,
    string deviceId,
    [DurableClient] IDurableEntityClient client,
    ILogger logger)
{
    var requestBody = await new StreamReader(req.Body).ReadToEndAsync();
    var telemetry =
JsonConvert.DeserializeObject<DeviceTelemetry>(requestBody);
    telemetry.DeviceId = deviceId;

    var entityId = await
_entityfactory.GetEntityIdAsync(telemetry.DeviceId, telemetry.Type,
default);

    await client.SignalEntityAsync<IDeviceEntity>(entityId, d =>
d.TelemetryReceived(telemetry));

    return new OkObjectResult(telemetry);
}
```

The **TelemetryDispatcher** component in our solution retrieves telemetry from the IoTHub, creates a **DeviceTelemetry** object, and sends it to the front-end layer using the **POST** API exposed by this function.

In the following figure, you can see the structure of the **DeviceTelemetry** class.

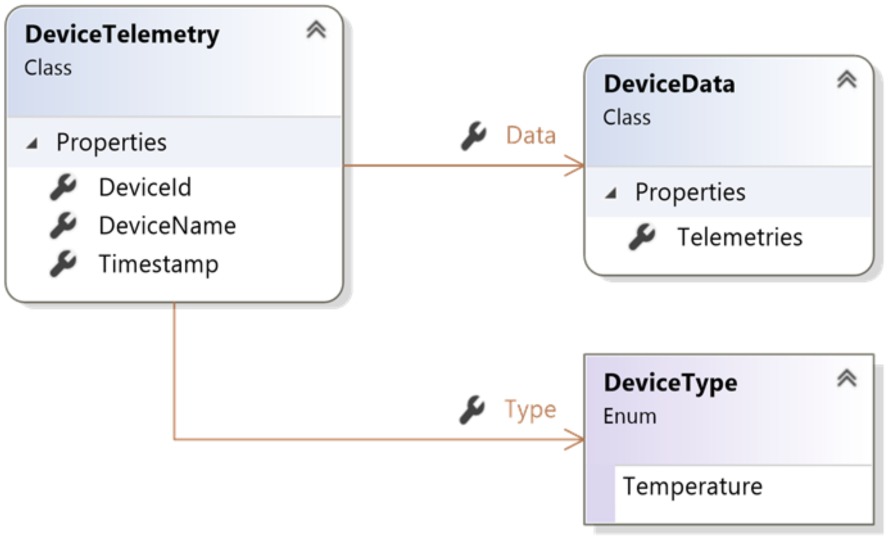


*Figure 33: DeviceTelemetry class structure*

You can add all the properties you need to enrich the information to send to the devices:

- **`DeviceId`**: This is the string that identifies a specific device. The **`SendTelemetryToDevice`** function uses it to create the **`IdentityId`** to signal data to the particular device.
- **`DeviceName`**: This is the name of the device.
- **`Timestamp`**: This is the date and time when the device sent the telemetry.
- **`DeviceData`**: This property contains the telemetries. It is a dictionary with a string key (the name of the telemetry, for example, **`temperature`** or **`humidity`**) and a double value (the value of the telemetry). Using a dictionary, you can have different devices with different telemetries managed in the same solution.
- **`DeviceType`**: This defines the type of the device. You have only a temperature device in the sample, but you can add other kinds of devices. The **`SendTelemetryToDevice`** uses this property (with the **`DeviceId`**) to generate the right **`IdentityId`** for the device to interact with.

The **`SendTelemetryToDevice`** client is easy, and its flow is the following:

1. Retrieves the **`DeviceTelemetry`** object from the request body.
2. Generates the **`EntityId`** using a factory class. The factory class (you can find its implementation in the GitHub repo) uses the **`DeviceId`** and the **`DeviceType`** values to instantiate an **`EntityId`** object. With this approach, you can add a new type of device simply adding a new value to the **`DeviceType`** enumeration, implementing the new device, and changing the behavior of the factory.
3. Signals the telemetry to the device using the **`EntityId`** created in step 2 and the operation **`TelemetryReceived`** exposed by the entity.

In this scenario, the signaling approach is the right choice because the client must send the telemetry to the device without waiting for the response. This way, it can immediately close and free its resources for the other telemetries sent by the **`TelemetryDispatcher`**.

The **GetDevices** client function allows you to search devices and retrieves information about them. The implementation code of this client is the following.

*Code Listing 47: GetDevices function*

```
public async Task<IActionResult> GetDevices(
  [HttpTrigger(AuthorizationLevel.Function, "get", Route = "devices")]
HttpRequest req,
  [DurableClient] IDurableEntityClient client)
{
    if (!Enum.TryParse(typeof(DeviceType), req.Query["deviceType"], true,
out var deviceType))
    {
        return new BadRequestResult();
    }

    var result = new List<DeviceInfoModel>();

    EntityQuery queryDefinition = new EntityQuery()
    {
        PageSize = 100,
        FetchState = true,

    };
    queryDefinition.EntityName = await
_entityfactory.GetEntityNameAsync((DeviceType)deviceType, default);

    do
    {
        EntityQueryResult queryResult = await
client.ListEntitiesAsync(queryDefinition, default);

        foreach (var item in queryResult.Entities)
        {
            DeviceInfoModel model = item.ToDeviceInfoModel();
            // If you want to add other filters to your method,
            // you can add them here before adding the model to the return
list.
            result.Add(model);
        }

        queryDefinition.ContinuationToken = queryResult.ContinuationToken;
    } while (queryDefinition.ContinuationToken != null);

    return new OkObjectResult(result);
}
```

The function expects a parameter in the query string of the request to define the device type. If this parameter is not present, the function returns an HTTP 400 (bad request) error.

The function code consists of two main steps.

The first step consists of creating an **EntityQuery** object to control the behavior of the **ListEntitiesAsync** method used later in the code. The **EntityQuery** class allows you to configure the page size of the list of entities for each request (**PageSize** property); to define if you want to retrieve only the device metadata or also the state (**FetchState** property); or to add a filter on the entity name (**EntityName** property).

If you don't set the **EntityName**, the **ListEntitiesAsync** method retrieves all the entities in the platform. Our sample generates the entity's name using a factory similar to the factory used in the **SendTelemetryToDevice** function to generate the **EntityId**.

The next logical step in the code is retrieving the entities data from the platform. For this purpose, you can use the **ListEntitiesAsync** method. Every time you call it, you retrieve a page of entities based on the setting you define in the **EntityQuery** class mentioned earlier. The return value of the **ListEntitiesAsync** method contains the page of results (property **Entities**) and the continuation token (**ContinuationToken** property).

If you want the next page of the query, you need to call the **ListEntitiesAsync** method again, passing **EntityQuery** with the continuation token retrieved in the previous call. You reach the last page when the continuation token returned by the **ListEntityAsync** method is null.

The **Entities** property is a list that contains the entity information. It includes the entity ID, the last operation timestamp, and the entity's status if you enable the **FetchState** property in the **EntityQuery**. This status is a JSON object stored in a **JToken** class.

In our example, we deserialize the JSON object to fill the list of the result to return to the client caller.

You can add other filters to your client functions (such as the device's name or the last operation timestamp) before adding the state to the return list.

The **GetDevice** client function allows you to retrieve information about a specific device, and its code is the following.

*Code Listing 48: GetDevice function*

```
[FunctionName(nameof(GetDevice))]
public async Task<IActionResult> GetDevice(
    [HttpTrigger(AuthorizationLevel.Function, "get", Route =
"devices/{deviceId}")] HttpRequest req,
    string deviceId,
    [DurableClient] IDurableEntityClient client)
{
    if (!Enum.TryParse(typeof(DeviceType), req.Query["deviceType"], true,
out var deviceType))
    {
        return new BadRequestObjectResult(deviceId);
    }

    EntityId entityId = await _entityfactory.GetEntityIdAsync(deviceId, 
(DeviceType)deviceType, default);

    EntityStateResponse<JObject> entity = await 
client.ReadEntityStateAsync<JObject>(entityId);
    if (entity.EntityExists)
    {
        var device = entity.EntityState.ToDeviceDetailModel();
        device.DeviceId = deviceId;
        return new OkObjectResult(device);
    }
    return new NotFoundObjectResult(deviceId);
}
```

The function's URL expects the ID of the device to be recovered and the **deviceType** field as a parameter in the query string. The code uses device ID and device type to generate the entity ID and uses this value to call the **ReadEntityStateAsync** method exposed by the **IDurableEntityClient**.

This method allows you to retrieve the entity's status and serialize it in the object used as generic. The method returns an **EntityStateResponse** object. This object contains the serialized status (**EntityState** property) and a Boolean property, which indicates if the entity exists or not (**EntityExists** property). If the entity exists, we create the response object and return it to the caller. Otherwise, the function returns a **NotFoundObjectResult** response (Not Found response, error code HTTP 404).

The **SetConfiguration** function allows you to configure a single device signaling the configuration object to the entity, as shown in the following code.

*Code Listing 49: SetConfiguration function*

```
[FunctionName(nameof(SetConfiguration))]
public async Task<IActionResult> SetConfiguration(
    [HttpTrigger(AuthorizationLevel.Function, "put", Route = 
"devices/{deviceId}/configuration")] HttpRequest req,
    string deviceId,
    [DurableClient] IDurableEntityClient client)
{
    if (!Enum.TryParse(typeof(DeviceType), req.Query["deviceType"], true, 
out var deviceType))
    {
        return new BadRequestObjectResult(deviceId);
    }

    EntityId entityId = await _entityfactory.GetEntityIdAsync(deviceId, 
(DeviceType)deviceType, default);

    var requestBody = await new StreamReader(req.Body).ReadToEndAsync();
    await client.SignalEntityAsync<IDeviceEntity>(entityId, d =>
d.SetConfiguration(requestBody));

    return new OkObjectResult(requestBody);
}
```

This function is similar to the **`SendTelemetryToDevice`** function: it retrieves the configuration object from the request body and signals it to the entity. The configuration object can be different for each device, and in this way, we can support different devices in the future.

Finally, the last client function, the **`GetDeviceNotifications`** function, retrieves all the devices' alert notifications. It is like the **`GetDevices`** function: it calls the **`ListEntitiesAsync`** method, but this time, the **`EntityName`** property of the **`EntityQuery`** object is always the same, and it is the name of the entity that stores the notifications for each device.

*Code Listing 50: GetDeviceNotifications function*

```
[FunctionName(nameof(GetDeviceNotifications))]
public async Task<IActionResult> GetDeviceNotifications(
  [HttpTrigger(AuthorizationLevel.Function, "get", Route =
"notifications")] HttpRequest req,
  [DurableClient] IDurableEntityClient client)
{
    var result = new List<JObject>();

    EntityQuery queryDefinition = new EntityQuery()
    {
        PageSize = 100,
        FetchState = true,
        EntityName = nameof(DeviceNotificationsEntity)
    };

    do
    {
        EntityQueryResult queryResult = await
client.ListEntitiesAsync(queryDefinition, default);

        foreach (var item in queryResult.Entities)
        {
            result.Add(item.State as JObject);
        }

        queryDefinition.ContinuationToken = queryResult.ContinuationToken;
    } while (queryDefinition.ContinuationToken != null);

    return new OkObjectResult(result);
}
```

## Stateful layer

The stateful layer contains the entities of our solution.

We have two kinds of entities:

- **`DeviceEntity`**: These entities implement the **`IDeviceInterface`** and represent the physical device of our IoT platform. In the example, we implement only one device, called **`TemperatureDeviceEntity`**, but you can extend the platform by adding other implementations of the same interface.
- **`DeviceNotificationEntity`**: Every time a device needs to send a notification, it uses one of these classes to store the event before calling an external system using an orchestrator.

The following figure shows the definition of the **`IDeviceInterface`** interface.

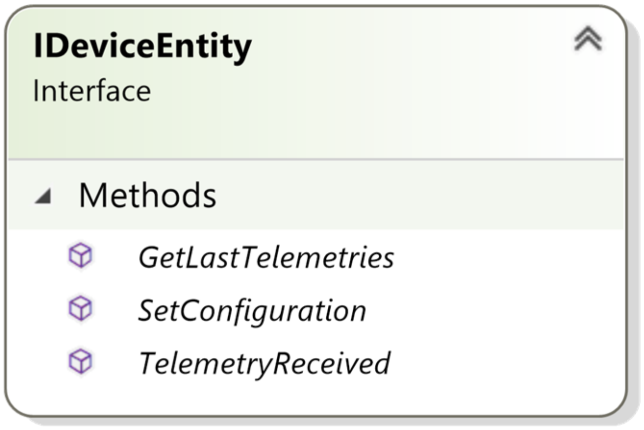


*Figure 34: The IDeviceEntity interface*

The interface exposes the following methods:

- **`TelemetryReceived`**: Allows you to send telemetry to a device.
- **`SetConfiguration`**: Allows you to change the device configuration (for example, the thresholds for notification) of a device.
- **`GetLastTelemetries`**: Allows you to retrieve the last telemetries stored in the device status.

The following snippet of code shows you an implementation of a device that manages telemetries containing a temperature value.

*Code Listing 51: The TemperatureDeviceEntity class*

```
[JsonObject(MemberSerialization.OptIn)]
public class TemperatureDeviceEntity : IDeviceEntity
{
    public class DeviceConfiguration
    {
      ...
    }

    private readonly ILogger logger;

    public TemperatureDeviceEntity(ILogger logger)
    {
        this.logger = logger;
        EntityConfig = new DeviceConfiguration();
    }

    #region [ State ]

    [JsonProperty("deviceType")]
    public string DeviceType {
       get => Models.DeviceType.Temperature.ToString();
       set { }
    }

    [JsonProperty("historyData")]
    public Dictionary<DateTimeOffset, DeviceData> HistoryData { get; set; }

    [JsonProperty("entityConfig")]
    public DeviceConfiguration EntityConfig { get; set; }

    [JsonProperty("deviceName")]
    public string DeviceName { get; set; }

    [JsonProperty("lastUpdate")]
    public DateTimeOffset LastUpdate { get; set; }

    [JsonProperty("lastData")]
    public DeviceData LastData { get; set; }

    [JsonProperty("temperatureHighNotificationFired")]
    public bool TemperatureHighNotificationFired { get; set; } = false;

    [JsonProperty("temperatureLowNotificationFired")]
    public bool TemperatureLowNotificationFired { get; set; } = false;

    #endregion [ State ]

    #region [ Behavior ]

    public void TelemetryReceived(DeviceTelemetry telemetry)
    {
      ...
    }
    public Task<IDictionary<DateTimeOffset, DeviceData>>
GetLastTelemetries(int numberOfTelemetries = 10)
    {
        ...
    }

    public void SetConfiguration(string config)
    {
        ...
    }
    #endregion [ Behavior ]

    #region [ Private Methods ]
      ...
    #endregion [ Private Methods ]

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

We will analyze the most important part of the code shortly, but you can find the whole class in the GitHub repo of this book.

The inner class, called **`DeviceConfiguration`**, contains the configuration you can set using the **`SetConfiguration`** method of the entity.

*Code Listing 52: The DeviceConfiguration class*

```
public class DeviceConfiguration
{
    [JsonProperty("historyRetention")]
    public TimeSpan HistoryRetention { get; set; } =
TimeSpan.FromMinutes(10);

    [JsonProperty("temperatureHighThreshold")]
    public double? TemperatureHighThreshold { get; set; }

    [JsonProperty("temperatureLowThreshold")]
    public double? TemperatureLowThreshold { get; set; }

    [JsonProperty("notificationNumber")]
    public string NotificationNumber { get; set; }

    public bool TemperatureHighAlertEnabled()
    {
        return TemperatureHighThreshold.HasValue;
    }

    public bool TemperatureLowAlertEnabled()
    {
        return TemperatureLowThreshold.HasValue;
    }
}
```

The properties of this class allow you to configure:

- The amount of telemetry to be stored by setting the time interval beyond which the telemetries are deleted (**`HistoryRetention`** property, by default 10 minutes).
- The thresholds for the temperature to be used to send notifications (the **`TemperatureHighThreshold`** and **`TemperatureLowThreshold`** properties).
- The telephone number to use to send notifications (the **`NotificationNumber`** property).

The following snippet of code shows you the implementation of the **`TelemetryReceived`** method.

*Code Listing 53: The TelemetryReceived method*

```
public void TelemetryReceived(DeviceTelemetry telemetry)
{
    if (HistoryData == null)
        HistoryData = new Dictionary<DateTimeOffset, DeviceData>();

    DeviceName = telemetry.DeviceName;

    if (telemetry.Timestamp <
DateTimeOffset.Now.Subtract(EntityConfig.HistoryRetention))
        return;

    if (telemetry.Data != null)
    {
        HistoryData[telemetry.Timestamp] = telemetry.Data;

        if (LastUpdate < telemetry.Timestamp)
        {
            LastUpdate = telemetry.Timestamp;
            LastData = telemetry.Data;
        }

        ClearHistoryData();
        CheckAlert();
    }
}
```

The method checks if the telemetry received by the caller must be stored (if it is older than the persistence interval set in the configuration, then it is discarded) and stores it in the **`HistoryData`** dictionary.

Finally, the method removes the expired telemetries (**`ClearHistoryData`** method) and checks if a notification must call (**`CheckAlert`** method).

If the device must throw a notification, it uses a method called **`SendAlert`**, which is shown in the following snippet of code.

*Code Listing 54: The SendAlert method*

```
private void SendAlert(double lastTemperature)
{
    var notification = new DeviceNotificationInfo()
    {
        Timestamp = DateTimeOffset.Now,
        DeviceId = Entity.Current.EntityKey,
        DeviceType = Entity.Current.EntityName
    };
    notification.Telemetries.Add("temperature", lastTemperature);
    notification.Metadata.Add("notificationNumber",
EntityConfig?.NotificationNumber);

    Entity.Current.SignalNotification(notification);

    if (!string.IsNullOrWhiteSpace(EntityConfig?.NotificationNumber))
    {
        TemperatureNotificationData temperatureAlert = new
TemperatureNotificationData()
        {
            DeviceName = DeviceName,
            NotificationNumber = EntityConfig.NotificationNumber,
            Temperature = lastTemperature
        };


Entity.Current.StartNewOrchestration("Alerts_SendTemperatureNotification",
temperatureAlert);
    }
}
```

This method signals to another durable entity, called **`DeviceNotificationEntity`**, the information about the notification. Every device has a notification entity that stores all its alerts, even if these didn't generate a real notification against an external service.

We implement the **`SignalNotification`** method to centralize the creation of the notification entity and the signal process. This method is an extension method of the **`IDurableEntityContext`** interface, and the code is shown in the following snippet.

*Code Listing 55: The SignalNotification method*

```
public static void SignalNotification(this IDurableEntityContext context,
    DeviceNotificationInfo notification)
{
    var notificationEntityId = new
EntityId(nameof(DeviceNotificationsEntity),
            $"{context.EntityName}|{context.EntityKey}");


Entity.Current.SignalEntity<IDeviceNotificationEntity>(notificationEntityId
,
        n => n.NotificationFired(notification));
}
```

The name of the notification entity used by the device to store its notifications is generated using the device entity name and the device entity ID. In this way, you have a pair of entities: one for the telemetries (the device), and one for all the notifications thrown by the device (the notification entity).

Using the signal approach, the device sends the information to the notification entity, but it doesn't wait. It continues immediately with the next instruction. This approach helps you in scalability.

The last part of the method starts; if you set the phone number for the notification, an orchestration to send the alarm. In this case, the orchestration start is asynchronous, and the device can immediately manage the next telemetry.

If you change the behavior of the orchestrator, you can change the way the notification will be sent. In the next section, we will analyze the orchestrator.

The following snippet of code shows the **`DeviceNotificationEntity`** implementation.

*Code Listing 56: The DeviceNotificationEntity class*

```
public class DeviceNotificationsEntity : IDeviceNotificationEntity
{
    private readonly ILogger logger;

    public DeviceNotificationsEntity(ILogger logger)
    {
        this.logger = logger;
    }

    #region [ State ]

    [JsonProperty("notifications")]
    public List<DeviceNotificationInfo> Notifications { get; set; }

    [JsonProperty("deviceType")]
    public string DeviceType { get; set; }

    [JsonProperty("deviceId")]
    public string DeviceId { get; set; }
    #endregion [ State ]

    #region [ Behavior ]

    public void NotificationFired(DeviceNotificationInfo notification)
    {
        if (notification == null)
            return;

        if (Notifications == null)
            Notifications = new List<DeviceNotificationInfo>();

        DeviceType = notification.DeviceType;
        DeviceId = notification.DeviceId;
        Notifications.Add(notification);
    }

    public Task PurgeAsync()
    {
        Notifications?.Clear();
        return Task.CompletedTask;
    }

    #endregion [ Behavior ]

    [FunctionName(nameof(DeviceNotificationsEntity))]
    public static Task Run([EntityTrigger] IDurableEntityContext ctx,
ILogger logger)
           => ctx.DispatchAsync<DeviceNotificationsEntity>(logger);

}
```

## Integration layer

The integration layer contains all the durable functions used for the interaction with external services.

When an entity device needs to interact with external services, it starts an orchestrator. The orchestrator manages the flow among all the activity functions used for interaction between our platform and the external services.

This way, as we saw for the notification approach between device entity and notification entity in the previous paragraph, it allows us to implement an async interaction between the entity and the orchestrator. We don't block the device while we wait for the orchestration to be completed.

In the following snippet, you can see the implementation of the **`SendTemperatureNotification`** orchestrator.

*Code Listing 57: The SendTemperatureNotification orchestrator*

```
[FunctionName("Alerts_SendTemperatureNotification")]
public async Task SendTemperatureNotification(
        [OrchestrationTrigger] IDurableOrchestrationContext context,
        ILogger logger)
{
    var notificationdata = context.GetInput<TemperatureNotificationData>();

    var smsData = new TwilioActivities.SmsData()
    {
        Number = notificationdata.NotificationNumber,
        Message = $"The temperature for device 
{notificationdata.DeviceName} is {notificationdata.Temperature}"
    };

    try
    {
        await context.CallActivityAsync("TwilioActivities_SendSMS",
smsData);
    }
    catch (System.Exception ex)
    {
        logger.LogError(ex, "Error during TwilioActivity invocation",
smsData);
    }
}
```

We implement an orchestrator even if we have only one activity in the notification process (as we do in the previous snippet of code).

It is useful if you remember that an entity can signal another entity or start an orchestrator, and you cannot call activity directly from the entity.

The **`SendTemperatureNotification`** orchestrator calls a single activity (the **`TwilioActivities_SendSMS`** activity) to send an SMS using Twilio. The code for that activity is shown in the following snippet.

*Code Listing 58: The SendMessageToTwilio activity*

```
[FunctionName("TwilioActivities_SendSMS")]
```

```
[return: TwilioSms(AccountSidSetting = "TwilioAccountSid", AuthTokenSetting
= "TwilioAuthToken")]
public CreateMessageOptions SendMessageToTwilio([ActivityTrigger]
IDurableActivityContext context,
    ILogger log)
{
    SmsData data = context.GetInput<SmsData>();

    log.LogInformation($"Sending message to : {data.Number}");

    var fromNumber =
this.configuration.GetValue<string>("TwilioFromNumber");

        var message = new CreateMessageOptions(new
PhoneNumber(data.Number))
    {
        From = new PhoneNumber(fromNumber),
        Body = data.Message
    };

    return message;
}
```

All the configurations for the Twilio account are stored in the configuration file, as shown in the following JSON.

*Code Listing 59: The configuration file for our function app*

```
{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "UseDevelopmentStorage=true",
    "FUNCTIONS_WORKER_RUNTIME": "dotnet",
    ...
    ...
    "TwilioAccountSid": "<twilio account SID>",
    "TwilioAuthToken": "<twilio auth token>",
    "TwilioFromNumber": "<twilio virtual number>"
  }
}
```

***Note: Twilio is a full solution for enterprise communications. One of the services provided by Twilio is SMS (Twilio Programmable Messaging). You can find the documentation here.***

## Summary

This concludes our tour of Azure Durable Functions.

Durable Functions is a powerful technology that you can use in different scenarios to solve various problems. As with all technologies, there are scenarios where it makes sense to use this technology, and others where it would be counterproductive. I hope this book gives you the tools to understand if Durable Functions is the right tool for you.

Good luck and *buona fortuna*!