Soluciones Azure

AZURE FUNCTIONS

Modelos de despliegue

FaaS (Functions-as-a-Service)

- El auge de la informática sin servidor es una de las innovaciones más importantes de la actualidad. Las tecnologías sin servidor, como Azure Functions o AWS Lambda, permiten a los desarrolladores centrarse por completo en escribir código. La lógica de negocios se ejecuta como funciones y no se necesita aprovisionar ni escalar manualmente la infraestructura. El proveedor de nube administra la infraestructura. La aplicación se escala horizontalmente o se reduce verticalmente de manera automática en función de la carga. Debido a esto, la creación de aplicaciones se vuelve más rápida y sencilla. Ejecutar dichas aplicaciones a menudo resulta más barato, porque solo se paga por los recursos informáticos que realmente usa el código.
- La arquitectura serverless habilita la ejecución de una aplicación mediante contenedores efímeros y sin estado; estos son creados en el momento en el que se produce un evento que dispare dicha aplicación. Contrariamente a lo que nos sugiere el término, serverless no significa «sin servidor», sino que éstos se usan como un elemento anónimo más de la infraestructura, apoyándose en las ventajas del cloud computing.
- Esta tecnología sin servidor apareció por primera vez en lo que se conoce como tecnologías de plataforma de aplicaciones como servicio (aPaaS), actualmente como FaaS.

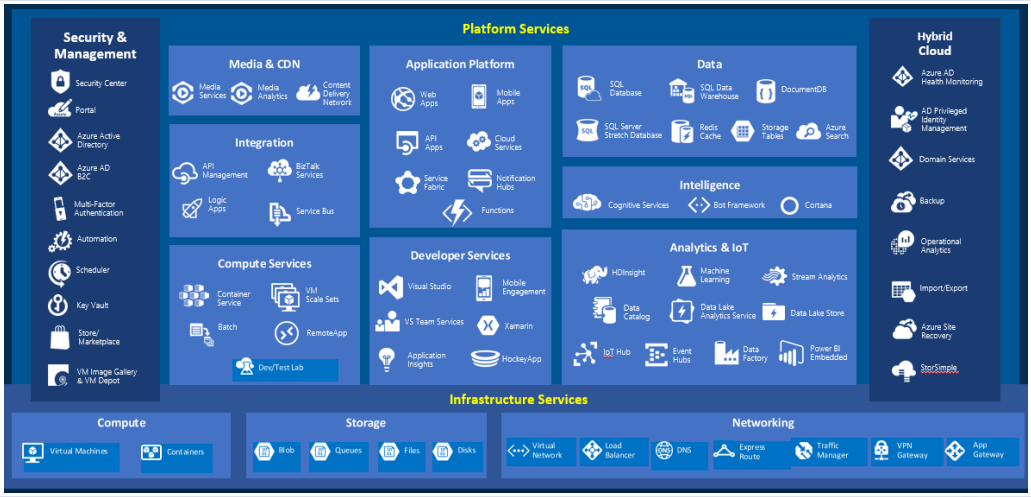
Azure Functions

- Azure Functions es una solución sin servidor que le permite escribir menos código, mantener menos infraestructura y ahorrar costos. En lugar de preocuparse por implementar y mantener servidores, la infraestructura en la nube proporciona todos los recursos actualizados necesarios para mantener las aplicaciones en ejecución.
- A menudo, se crean sistemas para que reaccionen a una serie de eventos críticos. Independientemente de si compila una API web, responde a cambios en una base de datos, procesa flujos de datos de IoT o incluso si administra colas de mensajes, cada aplicación necesita una forma de ejecutar código a medida que se producen estos eventos.
- Para ello, Azure Functions ofrece "proceso a petición" de dos maneras significativas:
  - Azure Functions permite implementar la lógica del sistema en bloques de código fácilmente disponibles. Estos bloques de código se denominan "funciones". Se pueden ejecutar distintas funciones cada vez que necesite responder a eventos críticos.
  - A medida que aumentan las solicitudes, Azure Functions satisface la demanda con tantos recursos e instancias de función como se necesiten, pero solo cuando sea necesario. A medida que disminuyan las solicitudes, todos los recursos e instancias de la aplicación adicionales se descartarán automáticamente.

Vertientes

- La tecnología nube está englobada en tres grandes vertientes dependiendo de la manera en la que se comercializa:
  - IaaS: Infraestructura como servicio que el cliente alquila y es totalmente escalable dependiendo de las necesidades.
  - PaaS: Plataforma como servicio donde puedes contar con las herramientas adecuadas para el desarrollo en nube evitando sobrecargas de sistema.
  - SaaS: Software como servicio que permite el alquiler de licencias dependiendo de las necesidades de cada usuario.

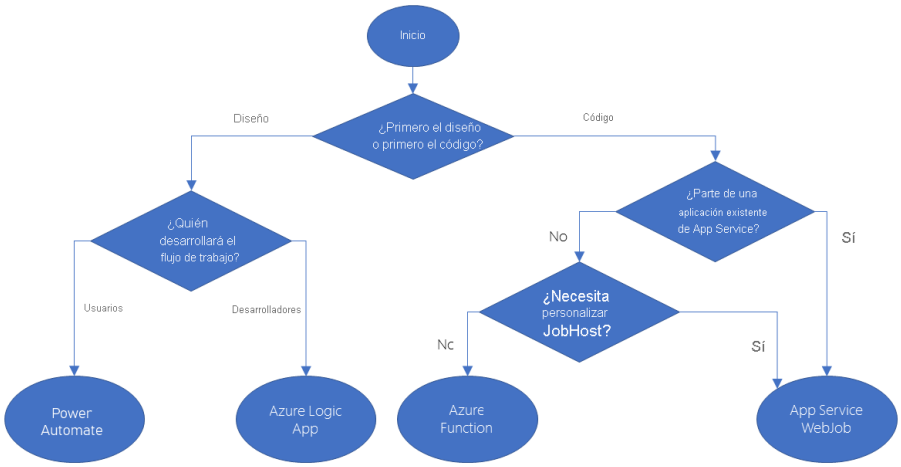
IaaS, PaaS, SaaS



Comparación serverless

- **Azure Functions** admite desencadenadores que son formas de iniciar la ejecución del código, y enlaces que son formas de simplificar la codificación para los datos de entrada y salida. Hay otros servicios de integración y automatización en Azure y todos ellos pueden resolver problemas de integración y automatizar procesos empresariales, y pueden definir entradas, acciones, condiciones y salidas.
- Azure Functions es un servicio de proceso sin servidor, mientras que **Azure Logic Apps** proporciona flujos de trabajo sin servidor. Ambos pueden crear orquestaciones complejas. Una orquestación es una colección de tareas que se ejecutan para realizar un proceso complejo. En Azure Functions, las orquestaciones se desarrollan mediante la escritura de código y el uso de la extensión Durable Functions, mientras que en Logic Apps, las orquestaciones se crean mediante una GUI o editando archivos de configuración. **Power Automate**, que se compila a partir de Logic Apps, permite a los usuarios realizar integraciones sencillas (por ejemplo, un proceso de aprobación en una biblioteca de documentos de SharePoint) sin pasar por los desarrolladores o el departamento de TI.
- Al igual que Azure Functions, **Azure App Service WebJobs** es un servicio de integración de tipo código primero que está diseñado para desarrolladores. Ambos se basan en Azure App Service y admiten características como la integración del control de código fuente, la autenticación y la supervisión con integración de Application Insights. Azure Functions se basa en el SDK de WebJobs, pero admite escalado automático, desarrollo y pruebas en el explorador, pago por uso e integración con Logic Apps.
- **Azure Event grid** es un servicio que habilita un modelo de publicador/suscriptor a escala web, que permite generar y consumir eventos desde dentro y fuera de Azure para coordinar eventos en toda la empresa. Los controladores de eventos se pueden implementar con Azure Functions.

Cómo se elige un servicio



Componentes de la plataforma Azure serverless

Opciones de hospedaje y escalado

- El plan de hospedaje elegido determina los comportamientos siguientes:
  - Cómo se escala la aplicación de funciones.
  - Los recursos disponibles para cada instancia de aplicación de funciones.
  - Compatibilidad con funcionalidad avanzada, como la conectividad con Azure Virtual Network.
- Los tres principales planes de hospedaje de Functions son:
  - Plan de consumo
  - Plan Premium
  - Plan dedicado
- En cualquier plan, Azure Functions requiere una cuenta de Azure Storage general que admita almacenamiento de Azure en blobs, colas, archivos y tablas, porque Functions se basa en Azure Storage para operaciones como la administración de desencadenadores y el registro de las ejecuciones de funciones.

Plan de consumo

- Escala de forma automática y se pagan los recursos de proceso solo cuando se ejecuten las funciones.
- En el plan de consumo, las instancias del host de Functions se agregan y quitan de forma dinámica según el número de eventos de entrada.
- Características:
  - Plan de hospedaje predeterminado.
  - Se paga solo cuando se ejecutan las funciones.
  - Escala de forma automática, incluso durante períodos de carga elevada.

Plan Premium

- Escala automáticamente en función de la demanda mediante trabajos preparados previamente que ejecutan aplicaciones sin ningún retraso después de estar inactivas, se ejecuta en instancias más eficaces y se conecta a redes virtuales.
- A elegir en las siguientes situaciones:
  - Las funciones se ejecutan de forma continua, o casi continua.
  - Hay un gran número de ejecuciones pequeñas y una factura de ejecución elevada, pero pocos GB por segundo en el plan de consumo.
  - Se necesita más opciones de CPU o memoria de las que proporciona el plan de consumo o características que no están disponibles, como la conectividad con red virtual.
  - El código debe ejecutarse durante más tiempo del máximo permitido en el plan de consumo.
  - Se quiere proporcionar una imagen personalizada de Linux en la que ejecutar sus funciones.

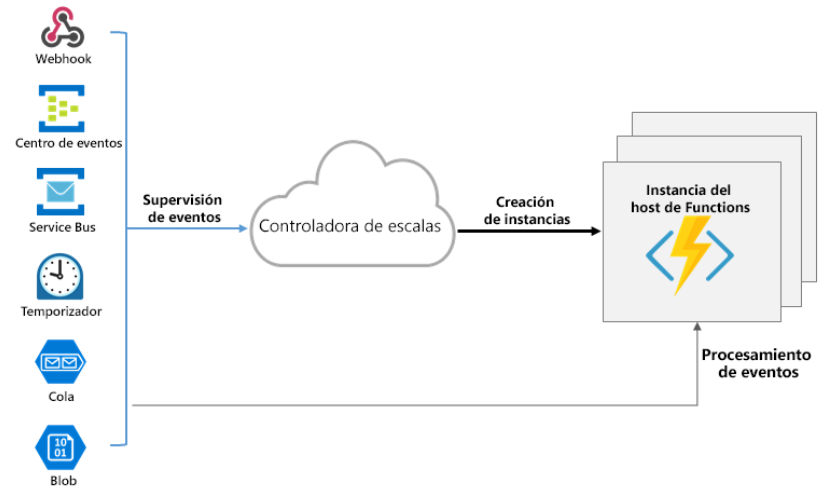
Plan dedicado

- Ejecuta las funciones en un plan de App Service a los Precios de App Service normales.
- Mejor para escenarios de ejecución prolongada en los que no se puede usar Durable Functions.
- A elegir en las siguientes situaciones:
  - Se dispone de máquinas virtuales infrautilizadas que ya ejecutan otras instancias de App Service.
  - Se requieren escalado y costos predictivos.

Escalado

- En los planes de consumo y Prémium, Azure Functions escala los recursos de CPU y memoria mediante la incorporación de instancias adicionales del host de Functions. El número de instancias se determina sobre el número de eventos que desencadenan una función.
- Cada instancia del host de Functions del plan de consumo tiene una limitación de 1.5 GB de memoria y una CPU. Una instancia del host es la aplicación de funciones completa, lo que significa que todas las funciones de una aplicación de funciones comparten recursos al mismo tiempo en una instancia y escala determinadas. Las aplicaciones de funciones que comparten el mismo plan de consumo se escalan de manera independiente. En el plan Prémium, el tamaño del plan determina la memoria y la CPU disponibles para todas las aplicaciones de ese plan en esa instancia.
- Los archivos de código de función se almacenan en recursos compartidos de Azure Files en la cuenta de almacenamiento principal de la función. Al eliminarse la cuenta de almacenamiento principal de la aplicación de función, los archivos de código de función también se eliminan y no se pueden recuperar.
- Azure Functions usa un componente denominado controlador de escala para supervisar la tasa de eventos y determinar si se debe escalar o reducir horizontalmente. El controlador de escala usa la heurística para cada tipo de desencadenador. La unidad de escala de Azure Functions es la aplicación de funciones. Al escalar horizontalmente la aplicación de función, se asignan recursos adicionales para ejecutar varias instancias del host de Azure Functions. Por el contrario, si la demanda se reduce, el controlador de escala elimina instancias del host de la función. El número de instancias se "reduce horizontalmente" hasta cero cuando no se ejecuta ninguna función en la aplicación de funciones.

Escalado

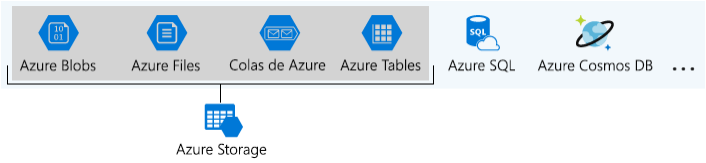


Comportamientos de escalado

- El escalado puede variar en función de varios factores, y realizarse de forma diferente según el desencadenador y el lenguaje seleccionados. Hay algunas complejidades de los comportamientos del escalado que hay que tener en cuenta:
  - Número máximo de instancias: Una aplicación de funciones única solo se escala horizontalmente hasta un máximo de 200 instancias. Una única instancia puede procesar más de un mensaje o solicitud a la vez, por lo que no hay un límite establecido en el número de ejecuciones simultáneas. Es posible restringir el número máximo de instancias que una aplicación usa para realizar el escalado horizontal.
  - Nueva tasa de instancias: En el caso de los desencadenadores HTTP, solo se asignan nuevas instancias como máximo una vez cada segundo. Para los desencadenadores que no son HTTP, solo se asignan nuevas instancias como máximo una vez cada 30 segundos.

Azure Storage

- Azure proporciona muchas maneras de almacenar los datos, incluidas varias opciones de base de datos, como Azure SQL Database, Azure Cosmos DB y Azure Table Storage. Azure ofrece varias maneras de almacenar y enviar mensajes, como Azure Queues y Event Hubs. Incluso puede almacenar archivos dinámicos mediante servicios como Azure Files y Azure Blobs.
- Azure agrupa cuatro de estos servicios de datos bajo el nombre Azure Storage. Los cuatro servicios son: Azure Blobs, Azure Files, Azure Queues y Azure Tables. En la siguiente ilustración se muestran los elementos de Azure Storage.



Azure Storage

- Azure Blobs: un almacén de objetos que se puede escalar de forma masiva para datos de texto y binarios.
- Azure Files: recursos compartidos de archivos administrados para implementaciones locales y en la nube.
- Azure Queues: un almacén de mensajería para mensajería confiable entre componentes de aplicación.
- Azure Tables: un almacén NoSQL para el almacenamiento sin esquema de datos estructurados.
- Azure Elastic SAN (versión preliminar): una solución totalmente integrada que simplifica la implementación, el escalado, la administración y la configuración de una SAN en Azure.
- Azure Disks: volúmenes de almacenamiento en el nivel de bloque para máquinas virtuales de Azure.

Azurite

- El emulador de código abierto Azurite proporciona un entorno local gratuito para probar las aplicaciones de Azure Blob, Queue Storage y Table Storage.
  string storageConnectionString = Environment.GetEnvironmentVariable("AzureWebJobsStorage");
- Se puede usar el acceso directo
  string storageConnectionString = "UseDevelopmentStorage=true";
- La cadena de conexión completa de HTTP es:
  DefaultEndpointsProtocol=http;AccountName=devstoreaccount1;AccountKey=Eby8vdM02xNOcqFlqUwJPLlmEtlCDXJ1OUzFT50uSRZ6IFsuFq2UVErCz4I6tq/K1SZFPTOtr/KBHBeksoGMGw==;BlobEndpoint=http://127.0.0.1:10000/devstoreaccount1;QueueEndpoint=http://127.0.0.1:10001/devstoreaccount1;TableEndpoint=http://127.0.0.1:10002/devstoreaccount1;
- Para usar solo Blob service, la cadena de conexión HTTP es:
  DefaultEndpointsProtocol=http;AccountName=devstoreaccount1;AccountKey=Eby8vdM02xNOcqFlqUwJPLlmEtlCDXJ1OUzFT50uSRZ6IFsuFq2UVErCz4I6tq/K1SZFPTOtr/KBHBeksoGMGw==;BlobEndpoint=http://127.0.0.1:10000/devstoreaccount1;
- Para usar solo Queue service, la cadena de conexión HTTP es:
  DefaultEndpointsProtocol=http;AccountName=devstoreaccount1;AccountKey=Eby8vdM02xNOcqFlqUwJPLlmEtlCDXJ1OUzFT50uSRZ6IFsuFq2UVErCz4I6tq/K1SZFPTOtr/KBHBeksoGMGw==;QueueEndpoint=http://127.0.0.1:10001/devstoreaccount1;
- Para usar solo Table service, la cadena de conexión HTTP es:
  DefaultEndpointsProtocol=http;AccountName=devstoreaccount1;AccountKey=Eby8vdM02xNOcqFlqUwJPLlmEtlCDXJ1OUzFT50uSRZ6IFsuFq2UVErCz4I6tq/K1SZFPTOtr/KBHBeksoGMGw==;TableEndpoint=http://127.0.0.1:10002/devstoreaccount1;

Entornos de desarrollo y lenguajes

- Azure Functions permite usar el editor de código y las herramientas de desarrollo que se prefieran para crear y probar las funciones en un equipo local. Las funciones locales pueden conectarse a servicios de Azure en directo, y se pueden depurar las funciones en el equipo local con el entorno de tiempo de ejecución de Functions completo.
- La manera en la que se desarrollan las funciones en el equipo local depende del lenguaje y las herramientas que se prefiera.
- Los principales lenguajes soportados son: C# (class library o script), JavaScript o TypeScript (Node.js), F#, Java, Python y PowerShell.
- Los siguientes entornos admiten el desarrollo de Azure Functions:
  - Azure Portal: online desde el navegador, solo es compatible con las funciones de JavaScript, PowerShell, TypeScript y scripts de C#.
  - Visual Studio
  - Visual Studio Code
  - Maven, Gradle: solo Java, compatible con el desarrollo con Eclipse y IntelliJ IDEA.
  - Terminal y símbolo del sistema: Azure Functions Core Tools permite desarrollar y probar funciones en el equipo local desde el símbolo del sistema o terminal.

Eventos y mensajería

- Los diferentes servicios de Azure emiten eventos para notificar los diferentes sucesos que están ocurriendo en ellos (cambios en la base de datos o el almacenamiento, notificaciones, peticiones HTTP, …) para asociar acciones a ellos. Así mismo los agentes de mensajearía empresarial incluyen colas de mensajes y temas que se pueden publicar y a los que es posible suscribirse.
- Para facilitar la conexión del código a otros servicios, Functions implementa un conjunto de extensiones de enlace para conectarse a estos servicios. Los desencadenadores y enlaces se proporcionan para consumir y emitir datos más fácilmente.
- Puede haber casos en los que se necesite obtener más control sobre la conexión del servicio o sea más cómodo usar el SDK proporcionado por el servicio, para usar una instancia de cliente del SDK en la ejecución de la función para acceder al servicio como se haría sin usar funciones, pero se debe prestar atención al efecto de la escalabilidad y el rendimiento en las conexiones de cliente.

Organización de las funciones

- La aplicación de función proporciona un contexto de ejecución en Azure donde ejecutar las funciones. Como tal, es la unidad de implementación y administración de las funciones. Una aplicación de función se compone de una o varias funciones individuales que se administran, implementan y escalan conjuntamente. Todas las funciones de una aplicación de función comparten el mismo plan de precios, el mismo método de implementación y la misma versión en tiempo de ejecución. Una aplicación de función es como una forma de organizar y administrar las funciones de manera colectiva.
- La forma de agrupar las funciones en aplicaciones de funciones puede afectar al rendimiento, el escalado, la configuración, la implementación y la seguridad de la solución global.
- El código de todas las funciones de una aplicación de funciones está ubicado en una carpeta de proyecto raíz que contiene un archivo de configuración de host. El archivo host.json contiene configuraciones específicas del entorno de ejecución. La carpeta bin contiene paquetes y otros archivos de biblioteca que requiere la aplicación de funciones. Las estructuras de carpeta específicas necesarias para la aplicación de funciones dependen del lenguaje.

Organización de las funciones

- Hay varios principios de diseño que puede seguir al escribir código de función que ayudan con el rendimiento general y la disponibilidad de las funciones. Estos principios incluyen:
  - Evitar funciones de larga duración que puedan cancelarse por timeout.
  - Escribir funciones que no tengan estado.
  - Planear la comunicación entre funciones.
  - Escribir funciones con la capacidad de continuar a partir de un punto de error anterior durante la siguiente ejecución.
  - Los lenguajes compilados ReadyToRun mejoran el rendimiento del inicio en frío.
- Dado que en la informática en la nube es común la aparición de errores transitorios, debe usar un patrón de reintento al acceder a recursos basados en la nube. Muchos desencadenadores y enlaces ya implementan el reintento.

Desencadenadores

- Una aplicación de Azure Functions no funciona hasta que algo le indica que se ejecute.
- Los desencadenadores son lo que provocan que una función se ejecute. Un desencadenador define cómo se invoca una función y cada función debe tener exactamente un desencadenador. Los desencadenadores también pueden procesar entradas para pasar datos a las funciones.
- Entre los tipos de desencadenador admitidos por Azure Functions se incluyen:
  - Almacenamiento: puede escuchar eventos de bases de datos como Azure Cosmos DB, por ejemplo, cuando se inserta una nueva fila.
  - Eventos: Event Grid y Event Hubs generan eventos que pueden desencadenar el código.
  - HTTP: el código se puede desencadenar mediante solicitudes web y webhooks.
  - Colas: los mensajes de colas también se pueden procesar.
  - Temporizador: el código se puede invocar con un intervalo de tiempo determinado.

Enlaces

- Se usan enlaces para conectarse a orígenes de datos. Los enlaces son formas declarativa de simplificar la codificación de los datos de entrada y salida. Aunque se pueden usar SDK de clientes para conectarse a servicios desde el código de la función, Functions proporciona enlaces para simplificar esas conexiones. Básicamente, los enlaces son código de conexión que no es necesario escribir. Permiten integrarse con muchos servicios de Azure para resolver problemas de integración y automatizar procesos empresariales.
- Hay dos tipos de enlaces: de entrada y de salida.
  - Los enlaces de entrada se pueden usar para pasar datos a la función desde un origen de datos diferente al que desencadenó la función. Los datos de los enlaces se proporcionan a la función como parámetros.
  - Un enlace de salida proporciona una manera de escribir datos en el origen de datos, por ejemplo, colocando un mensaje en una cola o una nueva fila en una base de datos. Un enlace de salida se corresponde con los datos que la función devuelve o envía.
- Se pueden mezclar y asignar enlaces diferentes para satisfacer las necesidades. A diferencia de un desencadenador, los enlaces son opcionales y cada función puede tener uno o varios enlaces de entrada y de salida.
- Los desencadenadores y enlaces evitan codificar el acceso a otros servicios. La función recibe los datos en parámetros de función y envía datos mediante el valor devuelto de la función.

Desencadenadores y enlaces

Tipo	Desencadenador	Input	Output	Tipo	Desencadenador	Input	Output
Blob storage	✓	✓	✓	Queue storage	✓		✓
Azure Cosmos DB	✓	✓	✓	RabbitMQ	✓		✓
Azure SQL (preview)	✓	✓	✓	SendGrid			✓
Dapr	✓	✓	✓	Service Bus	✓		✓
Event Grid	✓		✓	SignalR	✓	✓	✓
Event Hubs	✓		✓	Table storage		✓	✓
HTTP & webhooks	✓		✓	Timer	✓		
IoT Hub	✓			Twilio			✓
Kafka	✓		✓				
Mobile Apps		✓	✓				
Notification Hubs			✓				

Función

- Una función contiene dos elementos importantes: el código, que se puede escribir en diversos lenguajes, y la configuración. Con los lenguajes compilados, .NET (class library) y Java, este archivo de configuración se genera automáticamente a partir de las anotaciones del código. Para los lenguajes de scripting se debe proporcionar el archivo function.json de configuración.
- El archivo function.json define el desencadenador de la función, los enlaces y otras opciones de configuración. Cada función tiene un solo desencadenador. Este archivo de configuración se usa en tiempo de ejecución para determinar los eventos que se supervisarán y cómo pasar datos y devolverlos al ejecutarse una función.
- Para los lenguajes que dependen de function.json, el portal proporciona una interfaz de usuario para agregar enlaces en la pestaña Integración. También se puede editar el archivo directamente en el portal, en la pestaña Código y prueba de la función.

Configuración

- Los tipos de desencadenadores admitidos son: Blob storage, Azure Cosmos DB, Azure SQL (preview), Dapr, Event Grid, Event Hubs, HTTP & webhooks, IoT Hub, Kafka, Queue storage, RabbitMQ, Service Bus, SignalR, Timer.
- Todos los desencadenadores y enlaces tienen una propiedad de dirección en el archivo function.json:
  - En el caso de los desencadenadores, esta propiedad siempre aparece como in
  - Los enlaces de entrada y de salida usan in y out
  - Algunos enlaces admiten la dirección especial inout.
- Cuando se usan atributos en una biblioteca de clases para configurar los desencadenadores y los enlaces, la dirección se proporciona en un constructor de atributos o se deduce del tipo de parámetro.
- El tipo de parámetro define el tipo de datos de entrada, las opciones para dataType son binary, stream y string.

Configuración: function.json

```
{
  "bindings": [
    {
      "type": "queueTrigger",
      "direction": "in",
      "name": "order",
      "queueName": "myqueue-items",
      "connection": "MY_STORAGE_ACCT_APP_SETTING"
    },
    {
      "type": "table",
      "direction": "out",
      "name": "$return",
      "tableName": "outTable",
      "connection": "MY_TABLE_STORAGE_ACCT_APP_SETTING"
    }
  ]
}
```

Código: C# (script)

```
#r "Newtonsoft.Json"

using Microsoft.Extensions.Logging;
using Newtonsoft.Json.Linq;

// From an incoming queue message that is a JSON object, add fields and write to Table storage
// The method return value creates a new row in Table Storage
public static Person Run(JObject order, ILogger log)
{
   return new Person() {
         PartitionKey = "Orders",
         RowKey = Guid.NewGuid().ToString(),
         Name = order["Name"].ToString(),
         MobileNumber = order["MobileNumber"].ToString() };
}

public class Person
{
   public string PartitionKey { get; set; }
   public string RowKey { get; set; }
   public string Name { get; set; }
   public string MobileNumber { get; set; }
}
```


Código y Configuración: C # (class library)

```
public static class QueueTriggerTableOutput
{
   [FunctionName("QueueTriggerTableOutput")]
   [return: Table("outTable", Connection = "MY_TABLE_STORAGE_ACCT_APP_SETTING")]
   public static Person Run(
      [QueueTrigger("myqueue-items", Connection = "MY_STORAGE_ACCT_APP_SETTING")]JObject order,
      ILogger log)
   {
      return new Person() {
            PartitionKey = "Orders",
            RowKey = Guid.NewGuid().ToString(),
            Name = order["Name"].ToString(),
            MobileNumber = order["MobileNumber"].ToString() };
   }
}

public class Person
{
   public string PartitionKey { get; set; }
   public string RowKey { get; set; }
   public string Name { get; set; }
   public string MobileNumber { get; set; }
}
```

Proyecto de biblioteca de clases de Functions

- En Visual Studio, la plantilla de proyecto de Azure Functions crea un proyecto de biblioteca de clases de C# que contiene los archivos siguientes:
  - host.json: almacena los valores de configuración que afectan a todas las funciones del proyecto cuando se ejecuta localmente o en Azure.
  - local.settings.json: almacena la configuración de la aplicación y las cadenas de conexión que se utilizan cuando se ejecuta localmente. Este archivo contiene información confidencial que no se publicará en la aplicación de función de Azure.
- El proceso de compilación crea un archivo function.json para cada función. Este archivo function.json no está pensado para que se pueda modificar directamente. No se puede cambiar la configuración del enlace ni deshabilitar la función mediante la edición de este archivo.

Métodos Function

- En una biblioteca de clases, una función es un método con un atributo FunctionName y un atributo desencadenador.
- El atributo FunctionName marca el método como punto de entrada de una función. El nombre debe ser único dentro de un proyecto, hasta 127 caracteres y debe empezar por una letra y solo puede incluir letras, números, _ y -. Las plantillas de proyecto suelen crear un método denominado Run, pero el nombre de método puede ser cualquier nombre de método de C# válido.
- El atributo desencadenador especifica el tipo de desencadenador y enlaza los datos de entrada a un parámetro del método.

```
[FunctionName("Function1")]
    public static async Task<IActionResult> Run(
     [HttpTrigger(AuthorizationLevel.Function, "get", "post", Route = null)] HttpRequest req,
     ILogger log) {
```

Parámetros de la firma del método

- La firma del método puede contener parámetros distintos del usado con el atributo desencadenador. Estos son algunos de los otros parámetros que puede incluir:
  - Enlaces de entrada y salida marcados como tales con atributos.
  - Un parámetro ILogger o TraceWriter (ILogger exclusivamente) para el registro de la traza.
  - Un parámetro CancellationToken para el token de cancelación.
  - Parámetros de expresiones de enlace para obtener metadatos de desencadenador.
- El orden de los parámetros en la signatura de función no es importante. Por ejemplo, puede colocar los parámetros de desencadenador antes o después de otros enlaces y puede colocar el parámetro de registrador antes o después de los parámetros de desencadenador o enlace.

Propiedades de los enlaces

- Hay tres propiedades obligatorias en todos los enlaces. Es posible que deba proporcionar propiedades adicionales según el tipo de enlace y almacenamiento que use.
  - Nombre: define el parámetro de función a través del cual se accede a los datos (por ejemplo, en un enlace de entrada de cola, este es el nombre del parámetro de función que recibe el contenido del mensaje).
  - Tipo: identifica el tipo de enlace (por ejemplo, el tipo de datos o servicios con los que se quiere interactuar).
  - Dirección: identifica la dirección en la que fluyen los datos (por ejemplo, si es un enlace de entrada o salida).
- Además, la mayoría de los tipos de enlace también necesitan una cuarta propiedad:
  - Connection: proporciona el nombre de una clave de configuración de aplicación que contiene la cadena de conexión. Los enlaces usan cadenas de conexión almacenadas en la configuración de la aplicación para mantener los secretos fuera del código de la función. Esto hace que el código más seguro y configurable.

```
{
    "name": "image",
    "type": "blob",
    "path": "sample-images/{filename}",
    "connection": "HeadshotStorageConnection",
    "direction": "in"
},
```

Patrones de expresiones de enlace

- Una de las características más eficaces de los desencadenadores y enlaces son las expresiones de enlace. En el archivo function.json o en las anotaciones se pueden usar expresiones que se resuelvan como valores procedentes de diversos orígenes. La mayoría de las expresiones se identifican encerrándolos entre llaves.

```
"path": "samples-workitems/{queueTrigger}",
"path": "testContainerName/{date}-{filetype}.csv",
```

- Las expresiones se inyectan con parámetros de la función:

```
[FunctionName("ResizeImage")]
public static void Run([BlobTrigger("sample-images/{filename}")] Stream image,
    [Blob("sample-images-sm/{filename}", FileAccess.Write)] Stream imageSmall,
    string filename,  ILogger log)
```

- Además de la carga de datos que proporciona un desencadenador (como el contenido del mensaje de la cola que desencadenó una función), muchos desencadenadores facilitan valores de metadatos adicionales. La expresión de enlace {rand-guid} crea un identificador único global y {DateTime} se resuelve como DateTime.UtcNow.

Patrones de expresiones de enlace

- Como procedimiento recomendado, los secretos y las cadenas de conexión deberían administrarse mediante los ajustes de la aplicación, en lugar de archivos de configuración. De este modo, se limita el acceso a estos secretos y resulta seguro almacenar archivos como function.json en repositorios de control de código fuente públicos.
- Los ajustes de la aplicación también son útiles cuando se desea cambiar la configuración en función del entorno.
- Las expresiones de enlace de configuración de aplicación se identifican de forma diferente de otras expresiones de enlace: se encierran entre símbolos de porcentaje en lugar de entre llaves.

```
"queueName": "%input_queue_name%",
[FunctionName("QueueTrigger")]
public static void Run([QueueTrigger("%input_queue_name%")]string myQueueItem,
    ILogger log)
```

Valor devuelto

- Una función puede tener cero o varios enlaces de salida definidos mediante parámetros de salida. Los valores asignados a los enlaces de salida se escriben cuando finaliza la función. Se pueden usar más de un enlace de salida en una función simplemente asignando valores a varios parámetros de salida.

```
[Queue("myqueue-items-destination")] out string myQueueItemCopy,
```

- En los lenguajes que tienen un valor devuelto, se puede vincular el enlace de salida de una función al valor devuelto:

```
[FunctionName("QueueTrigger")]
[return: Blob("output-container/{id}")]
public static Task<string> Run([QueueTrigger("inputqueue")]WorkItem input, ILogger log) {
   string json = string.Format("{{ \"id\": \"{0}\" }}", input.Id);
   log.LogInformation($"C# script processed queue message. Item={json}");
   return Task.FromResult(json);
}
```


Control de errores y Reintentos

- El control de los errores en Azure Functions es importante para ayudar a evitar la pérdida de datos, evitar eventos perdidos y para supervisar el estado de la aplicación. Los errores podrían deberse a cualquiera de las siguientes acciones:
  - Uso de desencadenadores y enlaces de Azure Functions integrados
  - Llamadas a las API de los servicios de Azure subyacentes
  - Llamadas a puntos de conexión REST
  - Llamadas a bibliotecas de cliente, paquetes o API de terceros
- Varias extensiones de enlaces de Functions proporcionan compatibilidad integrada con reintentos. Además, el entorno de ejecución permite definir directivas de reintento para las funciones desencadenadas por Timer, Kafka y Event Hubs. Hay dos tipos de reintentos disponibles para sus funciones:
  - Comportamientos de reintento integrados de extensiones de desencadenador individuales
  - Directivas de reintento proporcionadas por el runtime de Functions: indica al runtime que vuelva a ejecutar una ejecución errónea hasta que se complete correctamente o se alcance el número máximo de reintentos.

```
[FunctionName("EventHubTrigger")]
[FixedDelayRetry(5, "00:00:10")]
public static async Task Run([EventHubTrigger("myHub", Connection = "EventHubConnection")] …
```

Registro

- En el código de la función, se puede escribir en los registros de salida que aparecen como seguimientos de Application Insights. La manera recomendada de escribir en los registros es incluir un parámetro de tipo ILogger, que normalmente se denomina log. No se debe usa Console.Write para escribir los registros, ya que Application Insights no captura los datos.

```
[FunctionName("Function1")]
public static async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Function, "get", "post", Route = null)] HttpRequest req,
    ILogger log) {
        log.LogInformation("C# HTTP trigger function processed a request.");
```

- Los métodos LogDebug, LogTrace, LogInformation, LogWarning, LogError y LogCritical permiten escribir registros con la categoría Function.<YOUR_FUNCTION_NAME>.User.
- Hay una versión específica del Application Insights SDK en Functions que se puede usar para enviar datos de telemetría personalizados desde las funciones a Application Insights. Se puede utilizar el método de extensión LogMetric de ILogger para crear métricas personalizadas en Application Insights.

Conexiones e identidades

- El proyecto de función hace referencia a la información de conexión por nombre de su proveedor de configuración. No acepta directamente los detalles de conexión, lo que permite que se modifiquen en todos los entornos. Por ejemplo, una definición de desencadenador podría incluir una propiedad connection. Esta podría hacer referencia a una cadena de conexión, pero no puede establecer la cadena de conexión directamente en un objeto function.json. En su lugar, debe establecer connection en el nombre de una variable de entorno que contenga la cadena de conexión.
- El proveedor de configuración predeterminado utiliza variables de entorno. Estas podrían establecerse mediante la configuración de la aplicación de Azure Portal cuando se ejecutan en el servicio Azure Functions, o desde el archivo de configuración local cuando el desarrollo se realiza localmente.

```
System.Environment.GetEnvironmentVariable("AzureWebJobsStorage", EnvironmentVariableTarget.Process);
```

- Cuando el nombre de la conexión se resuelve en un solo valor exacto, el runtime identifica el valor como una cadena de conexión, que normalmente incluye un secreto. Los detalles de una cadena de conexión se definen mediante el servicio al que quiere conectarse. Sin embargo, un nombre de conexión también puede hacer referencia a una colección de varios elementos de configuración. Las variables de entorno se pueden tratar como una colección mediante un prefijo compartido que termina en doble carácter de subrayado __ (Storage1__serviceUri). A continuación, se puede hacer referencia al grupo.

Conexiones e identidades

- Algunas conexiones de Azure Functions están configuradas para usar una identidad en lugar de un secreto. La compatibilidad depende de la extensión que utiliza la conexión. En algunos casos, es posible que aún se necesite una cadena de conexión en Functions aunque el servicio al que se está conectando admita conexiones basadas en identidades.
- Cuando se hospeda en el servicio de Azure Functions, las conexiones basadas en identidades usan una identidad administrada. La identidad asignada por el sistema se usa de manera predeterminada, aunque se puede especificar una identidad asignada por el usuario con las propiedades credential y clientID. Cuando se ejecuta en otros contextos, como el desarrollo local, se usa la identidad del desarrollador en su lugar, aunque se puede personalizar mediante parámetros de conexión alternativos.
- Cualquier identidad que se utilice debe tener permisos para realizar las acciones previstas. Normalmente, esto se realiza mediante la asignación de un rol en RBAC de Azure o la especificación de la identidad en una directiva de acceso, en función del servicio al que se esté conectando. Es posible que el servicio de destino muestre algunos permisos que no son necesarios para todos los contextos. Siempre que sea posible, se debe respetar el principio de privilegios mínimos y conceder solo los privilegios necesarios a la identidad.

Conexiones HTTP

- Azure App Service proporciona la infraestructura de hospedaje para las aplicaciones de funciones. Los componentes de plataforma de App Service, incluidas las máquinas virtuales de Azure, el almacenamiento, las conexiones de red, las plataformas web y las características de administración e integración se protegen y refuerzan activamente.
- Functions permite usar claves para dificultar el acceso a los puntos de conexión de función HTTP durante el desarrollo. A menos que el nivel de acceso HTTP en una función desencadenada por HTTP se establezca en anonymous, las solicitudes deben incluir una clave de acceso de API. Hay dos ámbitos de acceso para las claves de nivel de función:
  - Function: estas claves se aplican solo a las funciones específicas en las que se definen. Cuando se usan como una clave de API, solo permiten el acceso a esa función.
  - Host: las claves con un ámbito de host se pueden usar para acceder a todas las funciones dentro de la aplicación de función. Cuando se usan como una clave de API, permiten el acceso a cualquier función dentro de la aplicación de función.
- La clave se puede incluir en una variable de cadena de consulta denominada code o como un encabezado HTTP x-functions-key.

  https://<APP_NAME>.azurewebsites.net/api/<FUNCTION_NAME>?code=<API_KEY>
- Aunque las claves de función pueden proporcionar cierta mitigación del acceso no deseado, la única manera de proteger realmente los puntos de conexión de la función es mediante la implementación de la autenticación positiva de los clientes que acceden a las funciones. Después se pueden tomar decisiones de autorización en función de la identidad. La plataforma App Service permite usar Azure Active Directory (AAD) y varios proveedores de identidades de terceros para autenticar a los clientes.

Desencadenador de temporizador

- Con un desencadenador de temporizador puede ejecutar una función de forma programada.
- Azure Functions usa la biblioteca NCronTab para interpretar las expresiones NCRONTAB. Una expresión NCRONTAB es similar a una expresión CRON, excepto en que incluye un sexto campo adicional al comienzo para usarlo con una precisión de segundos:
  {second} {minute} {hour} {day} {month} {day-of-week}
- Cada campo puede tener uno de los siguientes tipos de valores:
  - Todos los valores (*)
  - Un valor específico
  - Un intervalo (operador -)
  - Un conjunto de valores (operador ,)
  - Un valor de intervalo (operador /)
- Para especificar meses o días puede usar valores numéricos, nombres o abreviaturas de nombres:
  - Para los días, los valores numéricos van de 0 a 6, donde 0 comienza con el domingo.
  - Los nombres están en inglés y no distinguen mayúsculas de minúsculas.
  - Los nombres se pueden abreviar. La longitud recomendada para la abreviatura es de tres letras. Por ejemplo: Mon, Jan.

Desencadenador de temporizador

```
[FunctionName("TimerTrigger")]
[return: Queue("myqueue-items")]
public static string RunTimerTrigger([TimerTrigger("*/15 * * * * *")] TimerInfo myTimer, ILogger log) {
   if(myTimer.IsPastDue) {
      log.LogInformation("Timer is running late!");
   }
   log.LogInformation($"C# Timer trigger function executed at: {DateTime.Now}");
   return $"C# Timer trigger function executed at: {DateTime.Now}";
}

[FunctionName("QueueTrigger")]
public static void QueueTrigger( [QueueTrigger("myqueue-items")] string myQueueItem, ILogger log) {
   log.LogInformation($"C# function processed: {myQueueItem}");
   Console.WriteLine(myQueueItem);
}
```

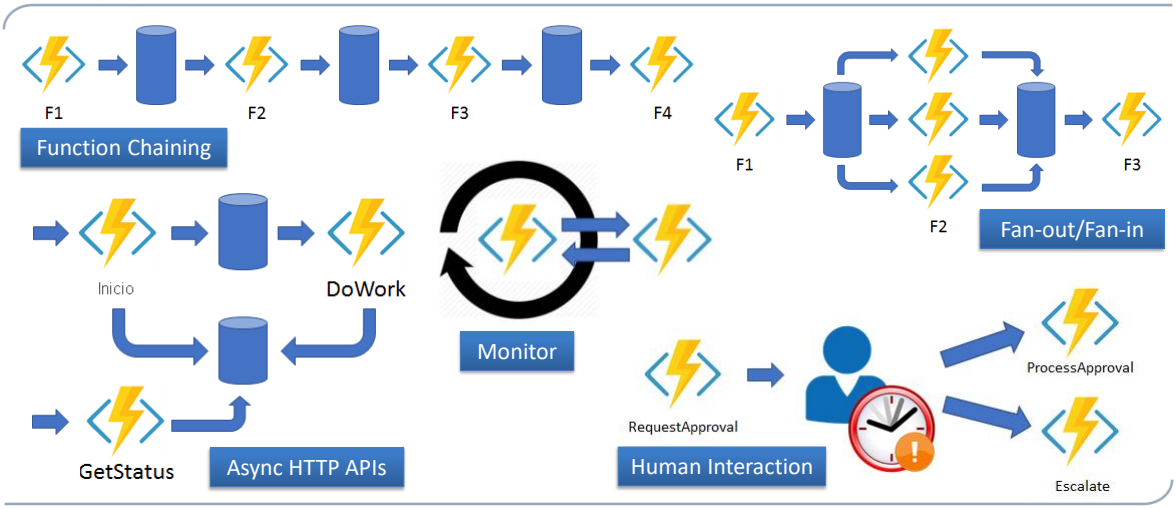
Flujos de trabajo

- Hay procesos simples que se resuelven con una simple tarea (función) y procesos complejos que requieren un flujo de trabajo: un conjunto de tareas síncronas/asíncronas, secuenciales/paralelas o con/sin intervención humana. La coordinación de los flujos de trabajo requieren el uso de patrones de orquestación y coreografía.
- En orquestación, contamos con un cerebro central para orientar y conducir el proceso, al igual que el director de una orquesta. La desventaja del enfoque de orquestación es que el orquestador se puede convertir en una autoridad de gobierno central demasiado fuerte y pasar a ser un punto central donde toda la lógica gira alrededor de él.
- Con coreografía, le informamos a cada parte del sistema de su trabajo y se les deja trabajar en los detalles, como los bailarines en un ballet que se encuentran en su camino y reaccionan ante otros a su alrededor. Podríamos tener una función emitiendo un mensaje asíncrono, las funciones interesados sólo se suscriben a esos eventos y reaccionan en consecuencia. Este enfoque es significativamente más desacoplado. Si algún nuevo servicio está interesado en recibir los mensajes, simplemente tiene que subscribirse a los eventos y hacer su trabajo cuando sea necesario.
- La desventaja de la coreografía es que la vista explícita del proceso de negocio del flujo de trabajo ahora sólo se refleja de manera implícita en el sistema. Esto implica que se necesita de trabajo adicional para asegurarse de que alguien pueda controlar y realizar un seguimiento de que hayan sucedido las cosas correctas. Para solucionar este problema, normalmente es necesario crear un sistema de monitoreo que refleje explícitamente el flujo de trabajo, pero que a su vez, haga un seguimiento de lo que cada una de las funciones realiza como entidad independiente que le permite ver excepciones mapeadas al flujo de proceso más explícito.

Patrones de aplicación

- **Encadenamiento de funciones**: una secuencia de funciones se ejecuta en un orden específico: la salida de una función se aplica a la entrada de otra función.
- **Distribución ramificada de salida y de entrada (Fan-out/Fan-in)**: se ejecutan en paralelo varias funciones y después se espera a que todas finalicen. A menudo se realiza algún trabajo de agregación en los resultados devueltos de las funciones.
- **Las API HTTP asíncronas**: soluciona el problema de coordinar el estado de las operaciones de larga duración con los clientes externos: un endpoint HTTP desencadena la acción de larga duración y devuelve un 202, el cliente se redirige al endpoint de estado al que sondea para consultar el estado y obtener el resultado cuando finalice la operación.
- **Supervisión**: se sondea mediante un desencadenador de temporizador hasta que se cumplen una condiciones específicas (patrón de arquitectura Monitor).
- **Interacción humana**: para procesos automatizados que implican algún tipo de interacción humana, ya que las personas no tienen la misma alta disponibilidad y capacidad de respuesta que los servicios en la nube. Los procesos automatizados pueden permitir esta interacción mediante el uso de tiempos de expiración y la lógica de compensación.

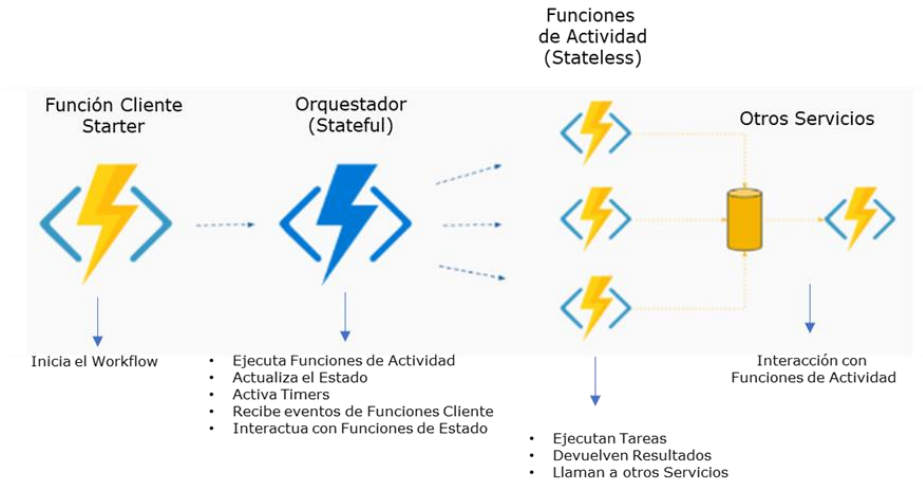
Patrones de aplicación



Durable Functions

- La extensión Durable Functions permite definir flujos de trabajo con estado mediante la escritura de funciones orquestador, actividad, entidad con estado y cliente con el modelo de programación de Azure Functions.
- En segundo plano, la extensión administra automáticamente el estado, los puntos de comprobación y los reinicios, lo que le permite centrarse en la lógica de negocios. Internamente se apoya en una central de tareas con almacenamientos, tablas y colas de mensajes y eventos para gestionar la orquestación.
- El caso de uso principal para Durable Functions es simplificar los requisitos de coordinación con estado complejos en las aplicaciones sin servidor.
- Durable Functions admite actualmente los siguientes lenguajes: C# (class library o script), JavaScript, F#, Python y PowerShell.

Durable Functions



Tipos de funciones

- Las **funciones orquestador** describen cómo se ejecutan las acciones y el orden en que se ejecutan las acciones. Las funciones orquestador describen la orquestación con código normal (C# o JavaScript), pero hay requisitos estrictos sobre cómo escribir el código. Una orquestación puede tener muchos tipos diferentes de acciones, como funciones de actividad, suborquestaciones, espera por eventos externos, HTTP y temporizadores. Las funciones orquestador también pueden interactuar con las funciones de entidad.
- Las **funciones de actividad** son la unidad básica de trabajo en una orquestación de función durable. Por ejemplo, podría crear una función orquestador para procesar un pedido. Las tareas implican comprobar el inventario, cobrar al cliente y crear un envío y cada una sería una función de actividad independiente. Estas funciones se pueden ejecutar en serie, en paralelo, o en una combinación de ambos métodos. Las funciones de actividad no tienen restricciones como el orquestador y pueden devolver datos a la función orquestador.

Tipos de funciones

- Las **funciones de entidad con estado** definen las operaciones de lectura y actualización de pequeños fragmentos del estado del proceso, a las que a menudo hacemos referencia como entidades duraderas. Al igual que las funciones orquestador, las de entidad son funciones con un tipo especial de desencadenador, el desencadenador de entidad.
- Las funciones orquestador y entidad con estado se desencadenan mediante sus enlaces y ambos desencadenadores funcionan reaccionando a los mensajes que se ponen en cola en un centro de tareas. La manera principal de entregar estos mensajes es mediante un enlace de cliente de orquestador, o un enlace de cliente de entidad, desde dentro de una función de cliente. Todas las funciones que no son de orquestador puede ser **funciones de cliente**. Lo que hace que una función sea una función de cliente es cómo usa el enlace de salida de cliente durable.

Central de tareas

- Una central de tareas en Durable Functions es una representación del estado actual de la aplicación en un almacenamiento duradero, incluido todo el trabajo pendiente. Mientras se ejecuta una aplicación de funciones, el progreso de las funciones de orquestación, actividad y entidad se almacena continuamente en la central de tareas. Esto garantiza que la aplicación pueda reanudar el procesamiento donde se dejó, si es necesario reiniciarlo después de que se detenga o interrumpa temporalmente por algún motivo. Además, permite que la aplicación de funciones escale dinámicamente los trabajos de proceso. Las funciones orquestador, actividad y entidad solo pueden interactuar entre sí si pertenecen a la misma central de tareas.
- Si varias aplicaciones de función comparten una cuenta de almacenamiento, se debe configurar cada una de ellas con un nombre de central de tareas independiente. Una cuenta de almacenamiento puede contener varias centrales de tareas. Esta restricción se suele aplicar también a otros proveedores de almacenamiento.
- Una central de tareas en Azure Storage consta de los siguientes recursos:
  - Una o más colas de control.
  - Una cola de elementos de trabajo.
  - Una tabla de historial.
  - Una tabla de instancias.
  - Un contenedor de almacenamiento que contiene uno o varios blobs de concesión.
  - Un contenedor de almacenamiento que contiene cargas de mensajes grandes, si procede.
  - Todos estos recursos se crean automáticamente en la cuenta de Azure Storage configurada cuando se ejecutan funciones orquestador, entidad o actividad o cuando se programa su ejecución.

Central de tareas

- Las centrales de tareas de Azure Storage se identifican mediante un nombre que se ajusta a estas reglas: solo caracteres alfanuméricos, comienza con una letra y entre 3 y 45 caracteres.
- El nombre de la central de tareas se declara en el archivo host.json

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

- El nombre es lo que diferencia una central de tareas de otra cuando hay varias de ellas en una cuenta de almacenamiento compartido. Si tiene varias aplicaciones de función que comparten una cuenta de almacenamiento, deberá configurar explícitamente nombres diferentes para cada central de tareas en el archivo host.json. En caso contrario, las diversas aplicaciones de funciones competirán entre sí por los mensajes, lo cual podría provocar un comportamiento indefinido, como, por ejemplo, orquestaciones que se "atascan" inesperadamente en el estado Pending o Running.

Función orquestador

- Las funciones orquestador tienen las siguientes características:
  - Las funciones orquestador definen flujos de trabajo de función con código de procedimiento. No se necesitan esquemas ni diseñadores.
  - Las funciones orquestador pueden llamar a otras funciones duraderas de forma sincrónica y asincrónica (se encolan). La salida de las funciones llamadas puede guardarse de forma confiable en variables locales.
  - Las funciones orquestador son duraderas y confiables. El progreso de la ejecución se controla automáticamente cuando la función está en espera o está procesándose. El estado local nunca se pierde si el proceso se recicla o se reinicia la máquina virtual.
  - Las funciones orquestador pueden ser de ejecución prolongada. La duración total de una instancia de orquestación puede ser de segundos, días o meses, o no terminar nunca.
- Cada instancia de una orquestación tiene un identificador (también conocido como identificador de instancia), un GUID generado automáticamente aunque pueden ser una cadena generada por el usuario. Cada identificador de instancia de orquestación debe ser único dentro de una central de tareas. El identificador de instancia de una orquestación es un parámetro necesario en la mayoría de las operaciones de administración de instancias.

Durable Task Framework

- Las funciones orquestador mantienen de forma confiable su estado de ejecución mediante el patrón de diseño event sourcing. En lugar de almacenar directamente el estado actual de una orquestación, Durable Task Framework utiliza un almacén solo de anexión para registrar la serie completa de acciones que la orquestación de función realiza.
- Durable Functions usa event sourcing de forma transparente. En un segundo plano, el operador await (C#) en una función orquestador devuelve el control del subproceso del orquestador al distribuidor de Durable Task Framework. El distribuidor, a continuación, confirma las nuevas acciones que la función orquestador programó (como llamar a una o más funciones secundarias o programar un temporizador durable) al almacenamiento. Esta acción de confirmación transparente se anexa al historial de ejecución de la instancia de orquestación. El historial se guarda en una tabla de almacenamiento. La acción de confirmación, a continuación, agrega mensajes a una cola para programar el trabajo real. En este momento, la función orquestador se puede descargar de la memoria.
- Cuando se proporciona más trabajo a una función de orquestación, el orquestador se reactiva y vuelve a ejecutar toda la función desde el principio para volver a generar el estado local. Durante esta reproducción, si el código intenta realizar una llamada a una función (o realizar cualquier otro trabajo asincrónico), Durable Task Framework consulta el historial de ejecución de la orquestación actual. Si descubre que la función de actividad ya se ha ejecutado y ofrece un resultado, reproduce el resultado de esa función y el código del orquestador sigue ejecutándose. La reproducción continúa hasta que finaliza el código de función o hasta que se programa un nuevo trabajo asincrónico. Para que el patrón de reproducción funcione correctamente y de forma confiable, el código de la función orquestador debe ser determinista.

Restricciones de código del orquestador

- Usar API deterministas, son aquellas que siempre devuelve el mismo valor con la misma entrada, con independencia de cuándo o con qué frecuencia se le llama.
- Las API que devuelven la fecha, la hora actual, un valor de GUID o UUID aleatorio son no deterministas, se debe usar la API equivalente de Durable Functions para obtener dichos valores, que permanece consistente a través de las repeticiones.
  - DateTime startTime = context.CurrentUtcDateTime;
  - Guid randomGuid = context.NewGuid();
- Usar una función de actividad para devolver números aleatorios a una orquestación dado que los valores devueltos por las funciones de actividad se guardan en el historial de orquestación y son siempre seguros para su reproducción.
- No usar variables de entorno y evitar el uso de variables estáticas en las funciones orquestador, sus valores pueden cambiar con el tiempo provocando un comportamiento no determinista.
- No debe usar ningún tipo de enlace, ni siquiera los enlaces del cliente de orquestación y del cliente de entidad, operaciones de E/S, llamadas de red salientes o iniciar operaciones asíncronas (excepto las definidas por el objeto de contexto del activador de orquestación) que se deben realizar en funciones de actividad.
- No crear o bloquear (sleep) subprocesos porque el orquestador se ejecuta en un solo subproceso.

Crear una función orquestador

- El desencadenador de orquestación permite crear funciones orquestador durables. Este desencadenador se ejecuta cuando se programa una nueva instancia de orquestación o cuando una ya existente recibe un evento. Entre los eventos que pueden desencadenar funciones orquestador se incluyen expiraciones de temporizadores de larga duración, respuestas de funciones de actividades y eventos desencadenados por clientes externos.

```
[FunctionName("Encadenamiento")]
public static async Task<List<string>> RunOrchestrator(
        [OrchestrationTrigger] IDurableOrchestrationContext context) {
```

- El enlace de desencadenador de orquestación admite entradas y salidas:
  - entradas: se pueden invocar con entradas a las que se accede a través del objeto de entrada de contexto y tienen que ser serializables con JSON.
  - salidas: admiten valores de salida, así como de entrada. El valor devuelto de la función se utiliza para asignar el valor de salida y tiene que ser serializable con JSON.

Crear una función orquestador

- La función recibe el contexto Durable de expone las operaciones que se pueden realizar. A través del contexto se puede acceder al parámetro de entrada:

```
string name = context.GetInput<string>();
```

- El valor devuelto por el método orquestador es el resultado de la orquestación.

```
return outputs;
```

- El método CallActivityAsync del contexto permite implementar el flujo de control mediante construcciones de código imperativas normales a funciones de actividad, devuelve una Task que se puede ejecutar secuencialmente.

```
var x = await context.CallActivityAsync<object>("F1", null);
var y = await context.CallActivityAsync<object>("F2", x);
```

- Las Task se pueden ejecutar en paralelo para realizar la distribución ramificada mediante una lista dinámica de las tareas. Se llama a Task.WhenAll para esperar a que todas las funciones llamadas finalicen.

```
var outputs = new List<string>();
var parallelTasks = new List<Task<string>>();
foreach(var city in new[] { "Tokyo", "Seattle", "London" })
   parallelTasks.Add(context.CallActivityAsync<string>(nameof(SayHello), city));
await Task.WhenAll(parallelTasks);
outputs = parallelTasks.Select(t => t.Result).ToList();
```

Función actividad

- El desencadenador de actividad permite crear las funciones de actividad que el orquestador ejecuta. Son funciones normales sin restricciones adicionales.

```
[FunctionName(nameof(SayGoodbye))]
public static string SayGoodbye([ActivityTrigger] IDurableActivityContext context, ILogger log) {
    string name = context.GetInput<string>();
```

- El enlace de desencadenador de actividad admite entradas y salidas como el desencadenador de orquestación:
  - entradas: se pueden invocar con entradas desde una función orquestador. Todas las entradas tienen que ser serializables con JSON.
  - salidas: admiten valores de salida, así como de entrada. El valor devuelto de la función se utiliza para asignar el valor de salida y tiene que ser serializable con JSON.
  - metadatos: las funciones de actividad de .NET pueden enlazar a un parámetro string instanceId para obtener el identificador de la instancia de la orquestación que llama.
- Los desencadenadores de actividad también admiten enlazar directamente con tipos serializables con JSON (incluidos los tipos primitivos), por lo que la función podría simplificarse:

```
[FunctionName(nameof(SayHello))]
public static string SayHello([ActivityTrigger] string name, ILogger log) {
```


Crear una función cliente de orquestación

- El enlace del cliente de orquestación permite escribir funciones de cliente que interactúan con las funciones orquestador y pueden:
  - Iniciarlas.
  - Consultar su estado.
  - Finalizarlas.
  - Enviarles eventos mientras se están ejecutando.
  - Purgar del historial de instancias.
- Las funciones de cliente pueden ser activadas por cualquier desencadenador de Functions, son funciones normales que se enlazan con DurableClient que proporciona el contexto de cliente de orquestación para acceder al API.
- Las orquestaciones y entidades se pueden invocar y administrar mediante solicitudes HTTP. Durable Functions tiene varias características que facilitan la incorporación de orquestaciones y entidades en flujos de trabajo HTTP.

Crear una función cliente de orquestación

- El método StartNewAsync del cliente activa la función orquestador y devuelve el identificador de la nueva instancia de orquestación:

```
[FunctionName("Encadenamiento_HttpStart")]
public static async Task<HttpResponseMessage> HttpStart(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post")] HttpRequestMessage req,
    [DurableClient] IDurableOrchestrationClient starter, ILogger log) {
    string instanceId = await starter.StartNewAsync("Encadenamiento", null);
    log.LogInformation("Started orchestration with ID = '{instanceId}'.", instanceId);
    return starter.CreateCheckStatusResponse(req, instanceId);
}
```

- Con CreateCheckStatusResponse se genera una estructura de webhook que se pueden usar para administrar la instancia de orquestación en destino. Se pueden devolver varios código de estado:
  - HTTP 202 (aceptado): la función orquestador especificada estaba programada para empezar a ejecutarse. El encabezado de respuesta Location contiene una dirección URL para sondear el estado de la orquestación.
  - HTTP 400 (solicitud incorrecta): la función orquestador especificada no existe, el identificador de instancia especificado no era válido o el contenido de la solicitud no era JSON válido.

Crear una función cliente de orquestación

- La respuesta HTTP está diseñada para ayudar a implementar la API HTTP asincrónicas de ejecución prolongada con Durable Functions, también conocida como Patrón de sondeo del consumidor.
- El flujo cliente/servidor funciona del siguiente modo:
  - El cliente emite una solicitud HTTP para iniciar un proceso de ejecución prolongada como una función orquestador.
  - El desencadenador HTTP de destino devuelve una respuesta HTTP 202 Accepted con un encabezado de ubicación que tiene el valor "statusQueryGetUri".
  - El cliente sondea la dirección URL en el encabezado de ubicación. El cliente continúa viendo las respuestas HTTP 202 con un encabezado de ubicación.
  - Cuando la instancia finaliza o se produce un error, el punto de conexión en el encabezado de ubicación devuelve HTTP 200.
- Este protocolo permite la coordinación de procesos de ejecución prolongada con clientes o servicios externos que pueden sondear un punto de conexión HTTP y seguir el encabezado de ubicación. Las implementaciones de cliente y de servidor de este patrón están integradas en las API HTTP de Durable Functions.

Crear una función cliente de orquestación

- La carga de respuesta para los casos HTTP 202 es un objeto JSON con los siguientes campos:
  - id: Identificador de la instancia de orquestación.
  - statusQueryGetUri: URL GET para obtener estado actual de la instancia de orquestación (&showHistory=true para ver el hitorial completo).
  - sendEventPostUri: URL POST para enviar eventos a la instancia de orquestación.
  - purgeHistoryDeleteUri: URL DELETE para la purga del historial de la instancia de orquestación.
  - suspendPostUri: URL POST para suspender la instancia de orquestación.
  - resumePostUri: URL POST para reanudar la instancia de orquestación.
  - terminatePostUri: URL POST para finalizar la instancia de orquestación.
  - rewindPostUri (versión preliminar): URL POST para reiniciar la instancia de orquestación.

Sub orquestaciones

- Las funciones orquestador pueden llamar a las funciones de actividad, pero también pueden llamar a otras funciones orquestador. Por ejemplo, puede crear una orquestación mayor a partir de una biblioteca de funciones orquestador. También puede ejecutar varias instancias de una función orquestador en paralelo.
- Las funciones de suborquestador se comportan como funciones de actividad desde la perspectiva del llamador. Pueden devolver un valor, producir una excepción y ser esperadas por la función orquestador primaria.
- Las funciones orquestador, cuando se invocan desde otro orquestador, se consideran funciones de suborquestador.
- Una función orquestador puede llamar a otra función orquestador mediante la API "call-sub-orchestrator".

```
outputs.AddRange(await context.CallSubOrchestratorAsync<List<string>>("Paralelismo",
        null));
```

Llamada a puntos de conexión HTTP

- No se permite que las funciones orquestador realicen E/S. La solución alternativa habitual para esta limitación es ajustar cualquier código que necesite realizar operaciones de E/S en una función de actividad. Las orquestaciones que interactúan con sistemas externos usan con frecuencia funciones de actividad que realizan las llamadas HTTP y devuelven el resultado a la orquestación.
- Para simplificar este patrón común, las funciones orquestador pueden usar el método CallHttpAsync para invocar directamente a las API HTTP.

```
[FunctionName("CheckSiteAvailable")]
public static async Task CheckSiteAvailable([OrchestrationTrigger] IDurableOrchestrationContext context) {
    Uri url = context.GetInput<Uri>();
    DurableHttpResponse response = await context.CallHttpAsync(HttpMethod.Get, url);
    if ((int)response.StatusCode == 404) {
        // handling of error codes goes here
    }
}
```

- Además de admitir patrones de solicitud y respuesta básicos, el método admite la autenticación de servicios externos mediante identidades administradas y el control automático de los patrones de sondeo asincrónicos HTTP 202.

Paso de varios parámetros

- No es posible pasar varios parámetros a una función de actividad directamente. Se recomienda realizar el paso en una matriz de objetos o usar objetos ValueTuples en .NET.

```
[FunctionName("GetCourseRecommendations")]
public static async Task<object> RunOrchestrator([OrchestrationTrigger] IDurableOrchestrationContext context) {
    string major = "ComputerScience";
    int universityYear = context.GetInput<int>();
    object courseRecommendations = await context.CallActivityAsync<object>("CourseRecommendations", (major, universityYear));
    return courseRecommendations;
}

[FunctionName("CourseRecommendations")]
public static async Task<object> Mapper([ActivityTrigger] IDurableActivityContext inputs) {
    (string Major, int UniversityYear) studentInfo = inputs.GetInput<(string, int)>();
    // retrieve and return course recommendations by major and university year
    return new {
        major = studentInfo.Major,
        universityYear = studentInfo.UniversityYear,
        recommendedCourses = new [] { "Introduction to .NET Programming", "Introduction to Linux", "Becoming an Entrepreneur" }
    };
}
```

Control del tiempo

- Las orquestaciones pueden programar temporizadores duraderos para implementar retrasos o configurar el control de tiempo de espera en acciones asincrónicas. Hay que usar los temporizadores durables del contexto de en lugar Durable Functions de Thread.Sleep y Task.Delay. Para crear un temporizador durable, se invoca el método CreateTimer del contexto. El método devuelve una tarea que se completa en una fecha y hora especificadas. La instancia de Durable Task Framework subyacente pone en cola un mensaje que se vuelve visible solo a la hora señalada.
- Para ejecutar cada día durante 10 días:

```
for (int i = 0; i < 10; i++) {
      DateTime deadline = context.CurrentUtcDateTime.Add(TimeSpan.FromDays(i + 1));
      await context.CreateTimer(deadline, CancellationToken.None);
      await context.CallActivityAsync("SendBillingEvent");
}
```


Control del tiempo

- Para implementar tiempos de expiración, Task.WhenAny crea una tarea que se completará cuando se haya completado cualquiera de las tareas proporcionadas:

```
DateTime deadline = context.CurrentUtcDateTime.Add(TimeSpan.FromSeconds(30));
using (var cts = new CancellationTokenSource()) {
   Task activityTask = context.CallActivityAsync("GetQuote");
   Task timeoutTask = context.CreateTimer(deadline, cts.Token);
   Task winner = await Task.WhenAny(activityTask, timeoutTask);
   if(winner == activityTask) { // cancel timeout
      cts.Cancel();
   }
   return winner == activityTask;
}
```

- Durable Task Framework no cambiará el estado de una orquestación a "Completed" hasta que todas las tareas pendientes se hayan completado o cancelado, por ello se usa cts.Cancel() para cancelar el temporizador.

Control del tiempo

- Las orquestaciones infinitas son funciones orquestador que nunca terminan, si se para la instancia continuará cuando se inicie la siguiente. Las funciones orquestador restablecen su estado mediante una llamada al método continue-as-new del contexto evitando la saturación del historial. Este método toma un parámetro serializable con JSON, que se convierte en la nueva entrada para la siguiente generación de la función orquestador y la instancia de orquestación se reinicia con el nuevo valor de entrada.
- Se mantiene el mismo identificador de instancia, pero se reinicia el historial de la función orquestadora. Para detener la ejecución hay que suspenderla y, opcionalmente, terminarla.

```
[FunctionName("Periodic_Loop")]
public static async Task RunPeriodic([OrchestrationTrigger] IDurableOrchestrationContext context) {
    Console.WriteLine(await context.CallActivityAsync<string>(nameof(DuracionVariable), 0));
    DateTime nextIteration = context.CurrentUtcDateTime.AddSeconds(5);
    await context.CreateTimer(nextIteration, CancellationToken.None);
    context.ContinueAsNew(null);
}
```


Eventos

- Las funciones orquestador tienen la capacidad de esperar y escuchar eventos externos que puede ser útil para controlar las interacciones humanas u otros desencadenadores externos.
- El método WaitForExternalEvent del contexto de orquestación permite que una función orquestador espere y escuche un evento externo de manera asíncrona. El método declara el nombre del evento y la forma de los datos que espera recibir.

```
bool approved = await context.WaitForExternalEvent<bool>("Approval");
if(approved) {
    // approval granted - do the approved action
} else {
    // approval denied - send a notification}
```

- El método RaiseEventAsync del contexto de cliente de orquestación envía el evento que espera WaitForExternalEvent. El método recibe la instancia, el nombre del evento y el valor del evento.

```
await client.RaiseEventAsync(instanceId, "Approval", true);
```

- Se puede generar un evento con una solicitud HTTP para una instancia de orquestación:

```
POST /runtime/webhooks/durabletask/instances/MyInstanceId/raiseEvent/Approval&code=XXX
Content-Type: application/json
"true"
```

Eventos

- Para implementar el patrón de correlación de eventos externos se pueden escuchar varios eventos al mismo tiempo:

```
var event1 = context.WaitForExternalEvent<float>("Event1");
var event2 = context.WaitForExternalEvent<bool>("Event2");
var event3 = context.WaitForExternalEvent<int>("Event3");
```

- Se puede esperar una de tres notificaciones de los eventos posibles:

```
var winner = await Task.WhenAny(event1, event2, event3);
```

- También es posible esperar a todos los eventos:

```
await Task.WhenAll(event1, event2, event3);
```

- La API "wait-for-external-event" espera indefinidamente alguna entrada.
- Internamente, RaiseEventAsync pone un mensaje en la cola de espera del evento de la función orquestador. Si la instancia no está esperando el nombre de evento especificado, el mensaje del evento se agrega a una cola en memoria. Si la instancia de orquestación inicia posteriormente la escucha de ese nombre de evento, se comprobará si hay mensajes de eventos en la cola.

Control de errores

- Las funciones orquestador pueden usar las características de control de errores como try/catch del lenguaje de programación.
- Cualquier excepción que se produce en una función de actividad se devuelve a la función orquestador en el búfer y se inicia como FunctionFailedException. Se puede escribir código de control y compensación de errores que se adapte a las necesidades en la función orquestador.
- Al llamar a funciones de actividad o funciones de suborquestación, se puede especificar una directiva de reintentos automáticos.

```
var retryOptions = new RetryOptions(firstRetryInterval: TimeSpan.FromSeconds(5),
    maxNumberOfAttempts: 3);
await context.CallActivityWithRetryAsync("UnstableFunction", retryOptions, null);
```

Función entidad con estado

- Las entidades duraderas proporcionan un mecanismo para realizar un seguimiento del estado de forma explícita dentro de las orquestaciones en lugar de hacerlo de forma implícita como parte del flujo de control. Son administradas por Durable Functions y funcionarán con cualquier opción de almacenamiento que se elija. Una de las ventajas de las entidades duraderas frente a la gestión de sus propios datos es que la simultaneidad se gestiona por usted. En lugar de manipular una entidad y almacenarla en una base de datos, las entidades duraderas se administran a través de operaciones que se envían con la garantía de que solo se ejecuta una sola operación en un momento dado para una entidad determinada y en el orden correcto. Esto evita que se produzcan condiciones de carrera.
- Las funciones de entidad con estado definen las operaciones de lectura y actualización de pequeños fragmentos del estado del proceso, también llamadas entidades duraderas. Permiten externalizar el estado de la orquestación, sin la interacción con un almacén de datos o una base de datos de back-end, e implementar patrones event aggregator. Las funciones de entidad proporcionan a los desarrolladores de aplicaciones sin servidor una manera cómoda de organizar el estado de la aplicación como una colección de entidades específicas.

Función entidad con estado

- Las entidades se comportan de forma algo parecida a pequeños servicios que se comunican mediante mensajes. Cada entidad tiene una identidad única y un estado interno (si existe). Al igual que los servicios u objetos, las entidades realizan operaciones cuando se les pide que lo hagan. Cuando se ejecuta una operación, es posible que actualice el estado interno de la entidad. También puede llamar a servicios externos y esperar una respuesta. Las entidades se comunican con otras entidades, orquestaciones y clientes mediante el uso de mensajes que se envían implícitamente a través de colas de confianza.
- Para evitar conflictos, se garantiza la ejecución en serie de todas las operaciones de una sola entidad, es decir, una después de otra.
- A las entidades se accede a través de un identificador único, el identificador de entidad. Un identificador de entidad es un par de cadenas que identifica de forma exclusiva una instancia de entidad y consta de un Nombre de entidad (un nombre que identifica el tipo de la entidad) y una Clave de entidad (cadena que identifica de forma única la entidad entre las demás entidades del mismo nombre, como un GUID).
- Para invocar una operación en una entidad, se especifica: Identificador de entidad, Nombre de la operación (cadena que especifica la operación que se va a realizar), Entrada de operación (un único parámetro de entrada opcional) y Tiempo programado (opcional para especificar el momento de ejecución). Las operaciones pueden devolver un valor de resultado o un resultado de error. Las orquestaciones que llamaron a la operación pueden observar este resultado o error. Si la instancia de entidad no existe, se creará automáticamente.
- Una operación de entidad también puede crear, leer, actualizar y eliminar el estado de la entidad. El estado de la entidad se conserva siempre de forma duradera en el almacenamiento.

Definición de entidades

- Actualmente se ofrecen dos API para definir las entidades:
  - La sintaxis basada en funciones es una interfaz de nivel inferior que representa a las entidades como funciones. Proporciona un control preciso sobre cómo se distribuyen las operaciones de entidad y cómo se administra el estado de la entidad.
  - La sintaxis basada en clases representa entidades y operaciones como clases y métodos. Esta sintaxis genera código fácilmente legible y permite invocar operaciones con comprobación de tipos mediante interfaces.
- La sintaxis basada en clases es simplemente una capa encima de la sintaxis basada en funciones, por lo que ambas variantes se pueden usar indistintamente en la misma aplicación. Las clases de entidad son POCO (objetos CLR estándar sin formato) que no requieren superclases, interfaces ni atributos especiales. Pero:
  - La clase debe poder construirse.
  - La clase debe ser código de JSON serializable.
- Además, cualquier método que se vaya a invocar como una operación debe cumplir requisitos adicionales:
  - debe tener como máximo un argumento y no debe tener ninguna sobrecarga ni argumentos genérico.
  - debe devolver Task o Task<T>.
  - los argumentos y los valores devueltos deben ser valores u objetos serializables.

Sintaxis basada en funciones

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

Sintaxis basada en clases

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


Acceso a entidades

- Se puede acceder a las entidades mediante una comunicación unidireccional o bidireccional. La terminología siguiente distingue las dos formas de comunicación:
  - La llamada (call) a una entidad utiliza la comunicación bidireccional (ida y vuelta). Envía un mensaje de operación a la entidad y, después, espera el mensaje de respuesta antes de continuar. El mensaje de respuesta puede proporcionar un valor de resultado o un resultado de error. El autor de la llamada observa este resultado o error.
  - La señalización (signal) de una entidad utiliza una comunicación unidireccional (desencadenar y olvidar). Envía un mensaje de operación pero no espera una respuesta. Aunque se garantiza la entrega del mensaje, el remitente no sabe cuándo y no puede observar ningún valor de resultado o error.
- Se puede acceder a las entidades a través de webhooks y desde el código de las funciones de cliente, orquestador o entidad con estado. No todos los contextos admiten todas las formas de comunicación:
  - En los clientes: ReadEntityState, SignalEntity, CleanEntityStorage y ListEntities.
  - En las orquestaciones: CallEntity y SignalEntity.
  - En las entidades, a otras entidades: SignalEntity.

Acceso a entidades por código

- Para crear el identificador de las entidades:
  ```
  var entityId = new EntityId(nameof(Counter), "steeps");
  ```
- Las funciones cliente pueden acceder a las entidades mediante el contexto cliente [DurableClient]:
  ```
  return client.SignalEntityAsync(entityId, "Add", int.Parse(input));
  ```
- Las funciones cliente también pueden consultar el estado de una entidad:
  ```
  var stateResponse = await client.ReadEntityStateAsync<int>(entityId);
  return new OkObjectResult(stateResponse.EntityState);
  ```
- Las funciones orquestador pueden acceder a las entidades mediante el contexto de orquestación:
  ```
  int currentValue = await context.CallEntityAsync<int>(entityId, "Get");
  if(currentValue >= 10) {
     context.SignalEntity(entityId, "Reset");
  } else {
     context.SignalEntity(entityId, "Add", 1);
  }
  ```
- Las funciones entidad con estado pueden acceder otras entidades mediante el contexto de entidad:
  ```
  ctx.SignalEntity(new EntityId("MonitorEntity", ""), "milestone-reached", ctx.EntityKey);
  ```


Acceso a entidades con webhooks

- Para enumerar todas las entidades de la central de tareas:
  GET /runtime/webhooks/durabletask/entities
  GET /runtime/webhooks/durabletask/entities/*Counter*
- Para obtener el estado de la entidad:
  GET /runtime/webhooks/durabletask/entities/*Counter*/*steps*
  - HTTP 200 (OK): el estado serializado en JSON de la entidad como su contenido.
  - HTTP 404 (NOT FOUND): no se encontró la entidad especificada.
- Para enviar un mensaje unidireccional de operación a una entidad:
  POST /runtime/webhooks/durabletask/entities/*Counter*/*steps*?op=*Add*
  Content-Type: application/json

  5
  - HTTP 202 (ACCEPTED): la operación de la señal se aceptó para el procesamiento asincrónico.
  - HTTP 400 (BAD REQUEST): el contenido de la solicitud no era del tipo application/json, no tenía un valor JSON válido o no tenía un valor entityKey válido.
  - HTTP 404 (NOT FOUND): no se encontró el entityName especificado.

Publicar el proyecto en Azure

- Para publicar el proyecto, deberá tener una aplicación de funciones ubicada en la suscripción de Azure. Las aplicaciones de función se pueden crear directamente desde Visual Studio.
  1. En el Explorador de soluciones, haga clic con el botón derecho en el proyecto y seleccione Publicar. En Destino, seleccione Azure y, luego, Siguiente. Seleccione la opción Aplicación de funciones de Azure (Windows) en la pestaña Destino específico. Esta opción creará una aplicación de funciones que se ejecutará en Windows y, luego, Siguiente.
  2. En Instancia de la función, seleccione Crear una aplicación de Azure Functions... y definir la nueva instancia especificando los valores de configuración.
  3. Seleccione Crear para crear una aplicación de funciones y sus recursos relacionados en Azure. El estado de la creación del recurso se muestra en la parte inferior izquierda de la ventana.
  4. En la instancia de Functions, asegúrese de que Ejecutar desde el archivo de paquete esté activado. La aplicación de funciones se implementa con la implementación de un archivo zip y con el modo de ejecución desde el paquete habilitado. La implementación de archivo ZIP es el método de implementación recomendado para el proyecto de Functions, ya que se obtiene un mejor rendimiento.
  5. Seleccione Finalizar y, en la página Publicar, seleccione Publicar para implementar el paquete que contiene los archivos de proyecto en la nueva aplicación de funciones en Azure. Una vez finalizada la implementación, en la pestaña Publicar aparecerá la dirección URL raíz de la aplicación de funciones.
  6. En la pestaña Publicar, en la sección Hospedaje, elija Abrir en Azure Portal. Esto abre el recurso de Azure de aplicación de funciones nuevo en Azure Portal.

Configuración del proyecto

Configuración	Descripción
Nombre	Nombre que identifica de forma única la nueva aplicación de función. Acepte este nombre o escriba uno nuevo. Los caracteres válidos son a-z, 0-9 y -.
Suscripción	La suscripción de Azure que se va a usar. Acepte esta suscripción o seleccione una nueva en la lista desplegable.
Grupo de recursos	Nombre del grupo de recursos en el que quiere crear la aplicación de funciones. Seleccione un grupo de recursos existente en la lista desplegable o la opción Nuevo para crear un nuevo grupo de recursos.
Tipo de plan	Cuando publique el proyecto en una aplicación de funciones que se ejecute en un plan Consumo, solo pagará por las ejecuciones de la aplicación. Otros planes de hospedaje suponen costos más elevados.
Ubicación	Elija una ubicación en una región próxima a usted o a otros servicios a los que las funciones accedan.
Azure Storage	El runtime de Functions necesita una cuenta de almacenamiento de Azure. Seleccione Nueva para configurar una cuenta de almacenamiento de uso general. También puede elegir una cuenta existente que cumpla los requisitos de la cuenta de almacenamiento.