AZ-204 – Guía Paso a Paso: Laboratorios de Azure Functions (.NET 8, dotnet-isolated)

Este documento detalla, paso a paso, cómo realizar cada ejercicio práctico basado en los requerimientos simulados. Cada laboratorio incluye: implementación local, configuración, despliegue a Azure, monitoreo y logging con Application Insights, seguridad/autenticación, integración con servicios y validación.

Requisitos previos

• Cuenta de Azure activa (puede ser Free Tier).

• Azure CLI actualizado (az).

• Azure Functions Core Tools v4.

• .NET 8 SDK.

• Visual Studio Code con extensiones: Azure Functions, Azure Storage, C#, REST Client (opcional).

• Azurite (opcional para emular Storage local): npm i -g azurite.

Variables comunes (CLI)

```
# Ajusta los valores a tu preferencia
LOCATION=eastus
RG=az204-rg
STORAGE=az204stor$RANDOM
FUNCAPP=az204-funcapp-$RANDOM
WORKSPACE=az204-la-$RANDOM
APPINSIGHTS=az204-ai-$RANDOM
# Crear grupo de recursos
az group create -n $RG -l $LOCATION
```

Ejercicio 1 – API de Consulta de Clientes (HTTP Trigger)

Escenario: E-commerce necesita una API serverless para consultar clientes por ID. Debe responder JSON, protegerse con clave de función o Entra ID, tener monitoreo y logs en Application Insights.

1) Crear proyecto local (.NET 8, dotnet-isolated)

```
mkdir http-clients && cd http-clients
func init --worker-runtime dotnet-isolated --target-framework net8.0
func new --name GetCustomer --template "HTTP trigger" --authlevel "function"
```

Estructura inicial de código (Program.cs)

```
using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

var host = new HostBuilder()
    .ConfigureFunctionsWorkerDefaults()
    .ConfigureLogging(logging => logging.AddFilter("Microsoft", LogLevel.Information))
    .Build();

host.Run();
```

Función GetCustomer.cs (ejemplo)

```
using System.Net;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Azure.Functions.Worker.Http;
using Microsoft.Extensions.Logging;

public class GetCustomer
{
    private readonly ILogger _logger;
    public GetCustomer(ILoggerFactory loggerFactory) => _logger = loggerFactory.CreateLogger<GetCustomer>(

    [Function("GetCustomer")]
    public HttpResponseData Run([HttpTrigger(AuthorizationLevel.Function, "get", Route = "customers/{id}")
    {
        _logger.LogInformation("Solicitud para cliente {Id}", id);

        // Simular acceso a datos (mock)
        var response = req.CreateResponse(HttpStatusCode.OK);
        response.Headers.Add("Content-Type", "application/json; charset=utf-8");
        var body = $"{{\"id\":\"{id}\",\"name\":\"Cliente Demo\",\"email\":\"cliente.demo@example.com\",\'
        response.WriteString(body);
        return response;
    }
}
```

2) Configuración local y pruebas

```
# Ejecutar local
func start

# Probar (reemplaza la key por la mostrada en consola si authlevel=function)
curl "http://localhost:7071/api/customers/42?code=<FUNCTION_KEY>"
```

3) Despliegue y configuración en Azure

```
# 3.1 Crear Storage y Function App (Consumption)
az storage account create -n $STORAGE -g $RG -l $LOCATION --sku Standard_LRS

# Workspace y App Insights (workspace-based)
az monitor log-analytics workspace create -g $RG -n $WORKSPACE -l $LOCATION
WORKSPACE_ID=$(az monitor log-analytics workspace show -g $RG -n $WORKSPACE --query customerId -o tsv)
WORKSPACE_KEY=$(az monitor log-analytics workspace get-shared-keys -g $RG -n $WORKSPACE --query primarySha
az monitor app-insights component create -g $RG -a $APPINSIGHTS -l $LOCATION --application-type web --work

# Crear Function App v4 (.NET 8, Linux, consumo)
az functionapp create -g $RG -n $FUNCAPP --storage-account $STORAGE   --consumption-plan-location $LOCATIO

# 3.2 Publicar
func azure functionapp publish $FUNCAPP
```

4) Seguridad y autenticación

Opción A: Mantener **Function Key** (rápido).

Opción B: Habilitar **Autenticación Entra ID (App Service Authentication)**:

```
# Habilitar autenticación (requiere provider configurado en Portal)
az webapp auth update -g $RG -n $FUNCAPP --enabled true --action LoginWithAzureActiveDirectory
```

Luego, desde Azure Portal → Function App → Authentication, agrega el proveedor Microsoft (Entra ID) con redirecciones predeterminadas y configura 'Require authentication' para todas las solicitudes.

5) Monitoreo y logging

```
# Ver métricas rápidas
az monitor metrics list --resource $(az functionapp show -g $RG -n $FUNCAPP --query id -o tsv)   --metric

# Logs en tiempo real
az webapp log tail -g $RG -n $FUNCAPP
```

6) Validación

Obtén la URL de la función desde el Portal o con func azure functionapp list-functions. Haz una llamada HTTP con la Function Key (o token Entra ID si habilitaste autenticación).

Ejercicio 2 – Proceso de Reportes Diarios (Timer Trigger)

Escenario: Un banco requiere generar cada día a las 6:00 AM un CSV con transacciones y subirlo a Blob Storage. Se debe registrar telemetría en Application Insights.

1) Proyecto y función Timer

```
mkdir timer-reports && cd timer-reports
func init --worker-runtime dotnet-isolated --target-framework net8.0
func new --name DailyReport --template "Timer trigger"
```

DailyReport.cs (cron 6 AM, output a Blob)

```
using System.Text;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.Logging;
using Azure.Storage.Blobs;
using System;
using System.Threading.Tasks;

public class DailyReport
{
    private readonly ILogger _logger;
    public DailyReport(ILoggerFactory loggerFactory) => _logger = loggerFactory.CreateLogger<DailyReport>(

    [Function("DailyReport")]
    public async Task Run([TimerTrigger("0 0 6 * * *")] TimerInfo myTimer)
    {
        _logger.LogInformation("Generando reporte diario a las: {Time}", DateTime.UtcNow);

        // Simular CSV
        var csv = "id,fecha,monto\n1,2025-01-01,100.0\n2,2025-01-01,55.5\n";
        var bytes = Encoding.UTF8.GetBytes(csv);

        // Subir a Blob
        var conn = Environment.GetEnvironmentVariable("AzureWebJobsStorage");
        var container = new BlobContainerClient(conn, "reports");
        await container.CreateIfNotExistsAsync();
        var blob = container.GetBlobClient($"report-{DateTime.UtcNow:yyyyMMdd}.csv");
        using var ms = new System.IO.MemoryStream(bytes);
        await blob.UploadAsync(ms, overwrite: true);

        _logger.LogInformation("Reporte subido: {BlobName}", blob.Name);
    }
}
```

2) Configuración local y pruebas

```
# Azurite (opcional, en otra terminal)
azurite

# local.settings.json usa: "AzureWebJobsStorage": "UseDevelopmentStorage=true"
func start
```

Para forzar ejecución local, cambia temporalmente el CRON a cada minuto: 0 * * * * *.

3) Despliegue y configuración

```
# Reutiliza $RG, $STORAGE, $FUNCAPP o crea nuevos
az storage account create -n $STORAGE -g $RG -l $LOCATION --sku Standard_LRS
```

```
az functionapp create -g $RG -n $FUNCAPP --storage-account $STORAGE   --consumption-plan-location $LOCATIO
func azure functionapp publish $FUNCAPP
```

4) Monitoreo y logging

Verifica en Application Insights → Logs (Kusto) con consultas como:

```
traces
| where message contains "Reporte subido"
| order by timestamp desc
```

5) Seguridad

El Timer no expone endpoint público, pero protege secretos usando Managed Identity y/o Key Vault para otras conexiones opcionales.

6) Validación

Confirma que el contenedor reports contiene el CSV del día.

Ejercicio 3 – Procesador de Imágenes (Blob Trigger + Output Binding)

Escenario: Al subir una imagen al contenedor incoming-images, la función procesa y guarda resultado en processed-images. Registrar en logs la ruta del archivo procesado.

1) Proyecto y función Blob Trigger

```
mkdir blob-processor && cd blob-processor
func init --worker-runtime dotnet-isolated --target-framework net8.0
func new --name ImageProcessor --template "Blob trigger"
```

ImageProcessor.cs (entrada y salida)

```
using System.Text;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.Logging;
using System.Threading.Tasks;
using Azure.Storage.Blobs;
using System;

public class ImageProcessor
{
    private readonly ILogger _logger;
    public ImageProcessor(ILoggerFactory loggerFactory) => _logger = loggerFactory.CreateLogger<ImageProce

    [Function("ImageProcessor")]
    public async Task Run([BlobTrigger("incoming-images/{name}", Connection = "AzureWebJobsStorage")] byte
                          string name)
    {
        _logger.LogInformation("Procesando imagen {Name}, tamaño: {Len}", name, input?.Length ?? 0);

        // Simular "marca de agua": anexar texto al contenido (demo)
        var processedBytes = Encoding.UTF8.GetBytes($"PROCESSED-{DateTime.UtcNow:o}-{name}");

        var conn = Environment.GetEnvironmentVariable("AzureWebJobsStorage");
        var container = new BlobContainerClient(conn, "processed-images");
        await container.CreateIfNotExistsAsync();
        var blob = container.GetBlobClient(name);
        using var ms = new System.IO.MemoryStream(processedBytes);
        await blob.UploadAsync(ms, overwrite: true);

        _logger.LogInformation("Imagen procesada guardada en: processed-images/{Name}", name);
    }
}
```

2) Configurar contenedores y probar localmente

```
# Crear contenedores en Storage (Az CLI o Portal)
az storage container create --name incoming-images --account-name $STORAGE --auth-mode login
az storage container create --name processed-images --account-name $STORAGE --auth-mode login

# Ejecutar local y probar
func start
# Sube un archivo de prueba al contenedor incoming-images (desde Portal o CLI)
```

3) Despliegue y configuración

```
# Desplegar
```

```
func azure functionapp publish $FUNCAPP

# Asegura que la app setting AzureWebJobsStorage del Function App apunta a tu Storage account
az functionapp config appsettings set -g $RG -n $FUNCAPP --settings "AzureWebJobsStorage=$(az storage acco
```

4) Monitoreo y logging

Revisa traces que contengan 'Procesando imagen' y 'Imagen procesada guardada' en Application Insights.

5) Seguridad

Restringe acceso al Storage con Private Endpoint o reglas de red. Usa Managed Identity para acceder al Storage con RBAC (Storage Blob Data Contributor).

6) Validación

Sube una imagen a incoming-images y verifica que aparezca en processed-images.

Ejercicio 4 – Procesamiento de Mensajes (Queue Trigger → Table Storage)

Escenario: Nuevas órdenes llegan a Azure Storage Queue orders. La función deserializa el JSON y guarda los datos en Table Storage. Registrar cada orden procesada.

1) Preparar recursos de cola y tabla

```
# Crear cola y tabla
az storage queue create --name orders --account-name $STORAGE --auth-mode login
az storage table create --name OrdersTable --account-name $STORAGE --auth-mode login
```

2) Proyecto y función Queue Trigger

```
mkdir queue-orders && cd queue-orders
func init --worker-runtime dotnet-isolated --target-framework net8.0
func new --name OrderProcessor --template "Queue trigger"
```

OrderProcessor.cs

```
using System.Text.Json;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.Logging;
using Azure.Data.Tables;
using System;
using System.Threading.Tasks;

public class Order
{
    public string OrderId { get; set; } = default!;
    public string CustomerId { get; set; } = default!;
    public double Amount { get; set; }
}

public class OrderEntity : ITableEntity
{
    public string PartitionKey { get; set; } = "Order";
    public string RowKey { get; set; } = Guid.NewGuid().ToString();
    public ETag ETag { get; set; }
    public DateTimeOffset? Timestamp { get; set; }
    public string OrderId { get; set; } = default!;
    public string CustomerId { get; set; } = default!;
    public double Amount { get; set; }
}

public class OrderProcessor
{
    private readonly ILogger _logger;
    public OrderProcessor(ILoggerFactory loggerFactory) => _logger = loggerFactory.CreateLogger<OrderProce

    [Function("OrderProcessor")]
    public async Task Run([QueueTrigger("orders", Connection = "AzureWebJobsStorage")] string message)
    {
        _logger.LogInformation("Mensaje recibido: {Message}", message);
        var order = JsonSerializer.Deserialize<Order>(message)!;

        var conn = Environment.GetEnvironmentVariable("AzureWebJobsStorage");
        var client = new TableClient(conn, "OrdersTable");
        await client.CreateIfNotExistsAsync();
        var entity = new OrderEntity
```

```
        {
            PartitionKey = "Order",
            RowKey = order.OrderId ?? Guid.NewGuid().ToString(),
            OrderId = order.OrderId,
            CustomerId = order.CustomerId,
            Amount = order.Amount
        };
        await client.AddEntityAsync(entity);

        _logger.LogInformation("Orden {OrderId} guardada en Table Storage.", order.OrderId);
    }
}
```

3) Probar localmente

```
func start
# Enviar un mensaje de prueba a la cola (reemplaza el contenido según tu caso)
az storage message put --queue-name orders --account-name $STORAGE --auth-mode login   --content '{"orderI
```

4) Despliegue y configuración

```
func azure functionapp publish $FUNCAPP
# Asegurar AzureWebJobsStorage configurado en el Function App (ver Ejercicio 3)
```

5) Monitoreo y logging

Consulta en Application Insights los logs que contengan 'Orden ... guardada en Table Storage'.

6) Seguridad

Usa Managed Identity con rol Storage Queue Data Reader y Storage Table Data Contributor si migras de cadenas de conexión a RBAC.

7) Validación

Verifica que el mensaje sea consumido y que la entidad aparezca en la tabla OrdersTable.

Ejercicio 5 – Durable Functions: Orquestación Fan-out/Fan-in

Escenario: Logística necesita calcular tiempos estimados de múltiples rutas en paralelo y consolidar el resultado total. Exponer un HTTP para iniciar la orquestación y requerir autenticación.

1) Proyecto Durable

```
mkdir durable-routes && cd durable-routes
func init --worker-runtime dotnet-isolated --target-framework net8.0
func new --name RoutesOrchestrator --template "Durable Functions orchestrator"
func new --name CalcRouteActivity --template "Durable Functions activity"
func new --name StartRoutes --template "Durable Functions HTTP starter"
```

RoutesOrchestrator.cs

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.Azure.Functions.Worker;
using Microsoft.DurableTask;
using Microsoft.Extensions.Logging;

public class RoutesOrchestrator
{
    [Function("RoutesOrchestrator")]
    public static async Task<int> RunOrchestrator([OrchestrationTrigger] TaskOrchestrationContext context)
    {
        var routes = new List<string> { "R1", "R2", "R3", "R4" };
        var tasks = new List<Task<int>>();
        foreach (var r in routes)
        {
            tasks.Add(context.CallActivityAsync<int>("CalcRouteActivity", r));
        }
        var results = await Task.WhenAll(tasks);
        var total = 0;
        foreach (var t in results) total += t;
        return total;
    }
}
```

CalcRouteActivity.cs

```
using System;
using System.Threading.Tasks;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Extensions.Logging;

public class CalcRouteActivity
{
    private readonly ILogger _logger;
    public CalcRouteActivity(ILoggerFactory loggerFactory) => _logger = loggerFactory.CreateLogger<CalcRou

    [Function("CalcRouteActivity")]
    public async Task<int> Run([ActivityTrigger] string route)
    {
        _logger.LogInformation("Calculando ruta {Route}", route);
        await Task.Delay(TimeSpan.FromMilliseconds(500)); // simular cálculo
        return new Random().Next(10, 60); // minutos estimados
    }
}
```

StartRoutes.cs (HTTP start)

```
using System.Net;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Azure.Functions.Worker.Http;
using Microsoft.DurableTask.Client;
using Microsoft.Extensions.Logging;
using System.Threading.Tasks;

public class StartRoutes
{
    private readonly ILogger _logger;
    public StartRoutes(ILoggerFactory loggerFactory) => _logger = loggerFactory.CreateLogger<StartRoutes>(

    [Function("StartRoutes")]
    public async Task<HttpResponseData> Run(
        [HttpTrigger(AuthorizationLevel.Function, "post", Route = "routes/start")] HttpRequestData req,
        [DurableClient] DurableTaskClient client)
    {
        var instanceId = await client.ScheduleNewOrchestrationInstanceAsync("RoutesOrchestrator");
        _logger.LogInformation("Orchestrator iniciado: {InstanceId}", instanceId);
        var response = req.CreateResponse(System.Net.HttpStatusCode.Accepted);
        response.WriteString($"InstanceId: {instanceId}");
        return response;
    }
}
```

2) Pruebas locales

```
func start
# Iniciar orquestación (reemplaza FUNCTION_KEY)
curl -X POST "http://localhost:7071/api/routes/start?code=<FUNCTION_KEY>"
```

3) Despliegue y autenticación

```
func azure functionapp publish $FUNCAPP

# Habilitar autenticación (App Service Authentication) y requerir inicio de sesión
az webapp auth update -g $RG -n $FUNCAPP --enabled true --action LoginWithAzureActiveDirectory
```

Luego, configura el proveedor Microsoft (Entra ID) desde el Portal. Exige autenticación para rutas HTTP.

4) Monitoreo y estado Durable

En el Portal: Function App → Functions → Durable Functions → Instances para ver estado. En Application Insights, filtra traces por 'Orchestrator iniciado' o 'Calculando ruta'.

5) Validación

Lanza varias ejecuciones y verifica el tiempo total devuelto por la orquestación.

Buenas prácticas generales

• Usa Managed Identity y Key Vault para secretos; evita cadenas de conexión embebidas.

• Activa Application Insights (workspace-based) y crea alertas métricas (latencia, errores, ejecuciones).

• Define Settings en Azure (Configuration) y usa local.settings.json para desarrollo local (no se sube a Git).

• Elige el plan correcto: Consumption (coste por ejecución), Premium (cold start reducido, VNET), Dedicated (cargas constantes).

• Automatiza despliegue con GitHub Actions o Azure DevOps usando func azure functionapp publish o acciones oficiales.

• Restringe red con Private Endpoints/VNET Integration si manipulas datos sensibles.

• Para triggers de Storage, valida reintentos y dead-letter donde aplique (p. ej., Service Bus DLQ).