# Lab: Implement task processing logic by using Azure Functions

# Student lab manual

## Lab scenario

Your company has built a desktop software tool that parses a local JavaScript Object Notation (JSON) file for its configuration settings. During its latest meeting, your team decided to reduce the number of files that are distributed with your application by serving your default configuration settings from a URL instead of from a local file. As the new developer on the team, you've been tasked with evaluating Microsoft Azure Functions as a solution to this problem.

## Objectives

After you complete this lab, you will be able to:

* Create a Functions app.
* Create various functions by using built-in triggers.
* Configure function app triggers and input integrations.

## Lab setup

* Estimated time: **45 minutes**

## Instructions

### Before you start

#### Sign in to the lab virtual machine

Ensure that you're signed in to your Windows 10 virtual machine (VM) by using the following credentials:

* Username: **Admin**
* Password: **Pa55w.rd**

#### Review the installed applications

Find the taskbar on your Windows 10 desktop. The taskbar contains the icons for the applications that you'll use in this lab:

* Microsoft Edge
* File Explorer
* Windows Terminal

### Exercise 1: Create Azure resources

#### Task 1: Open the Azure portal

1. Sign in to the Azure portal (<https://portal.azure.com>).
2. If this is your first time signing in to the Azure portal, you'll notice a dialog box offering a tour of the portal. Select **Get Started** to skip the tour.

#### Task 2: Create an Azure Storage account

1. Create a new storage account with the following details:
   * New resource group: **Serverless**
   * Name: **funcstor*[yourname]***
   * Location: **(US) East US**
   * Performance: **Standard**
   * Account kind: **StorageV2 (general purpose v2)**
   * Replication: **Locally-redundant storage (LRS)**
   * Access tier: **Hot**

* **Note**: Wait for Azure to finish creating the storage account before you move forward with the lab. You'll receive a notification when the account is created.

#### Task 3: Create a Functions app

1. Create a new function app with the following details:
   * Existing resource group: **Serverless**
   * App name: **funclogic*[yourname]***
   * Publish: **Code**
   * Runtime stack: **.NET Core**
   * Region: **East US**
   * Storage account: **funcstor*[yourname]***
   * Operating system: **Windows**
   * Plan: **Consumption**
   * Enable Application Insights: **No**

* **Note**: Wait for Azure to finish creating the function app before you move forward with the lab. You'll receive a notification when the app is created.

#### Review

In this exercise, you created all the resources that you'll use for this lab.

### Exercise 2: Create a function that's triggered by an HTTP request

#### Task 1: Create an HTTP-triggered function

1. Access the **funclogic*[yourname]*** function app that you created earlier in this lab.
2. Create a new function by using the following settings:
   * Development environment: **In-portal**
   * Template: **Webhook + API (HTTP Trigger)**
   * Name: **Echo**
   * Authorization level: **Anonymous**

#### Task 2: Write function code

1. In the function editor, delete the example **run.csx** function script.
2. Add the following **using** directives for libraries that will be referenced by the application:

* using Microsoft.AspNetCore.Mvc;  
  using System.Net;

1. Create a new **public static** method named **Run** that returns a variable of type **IActionResult** and that also takes in variables of type **HttpRequest** and **ILogger** as parameters named *req* and *log*:

* public static IActionResult Run(HttpRequest req, ILogger log)  
  {  
  }

1. Within the **Run** method, log a fixed message:

* log.LogInformation("Received a request");

1. Finally, echo the body of the HTTP request as the HTTP response:

* return new OkObjectResult(req.Body);

1. Save the updated function script.

#### Task 3: Test function run in the portal

1. Select **Logs** to observe the compilation results. The results should include a "Compilation succeeded" message.
2. Select **Run** to test the function.
3. Observe the results of the test run. The results should echo the original request body exactly.

#### Task 4: Get a base function URL

1. Access the **funclogic*[yourname]*** function app that you created earlier in this lab.
2. Copy the value of the **URL** text box. You'll use this value later in the lab.

#### Task 5: Test function run by using httprepl

1. Open the Windows Terminal application.
2. Start the **httprepl** tool, and then set the base Uniform Resource Identifier (URI) to the **URL** value that you copied earlier in this lab.

* httprepl <function-app-url>
* **Note**: For example, if your URL is **https://funclogicstudent.azurewebsites.net**, your command would be **httprepl https://funclogicstudent.azurewebsites.net**.

1. At the tool prompt, browse to the relative **api/echo** directory:

* cd api  
  cd echo

1. Run the **post** command sending in an HTTP request body set to a numeric value of **3** by using the **--content** option:

* post --content 3

1. Run the **post** command sending in an HTTP request body set to a numeric value of **5** by using the **--content** option:

* post --content 5

1. Run the **post** command sending in an HTTP request body set to a string value of **"Hello"** by using the **--content** option:

* post --content "Hello"

1. Run the **post** command sending in an HTTP request body set to a JSON value of **{"msg": "Successful"}** by using the **--content** option:

* post --content "{"msg": "Successful"}"

1. Exit the **httprepl** application:

* exit

1. Close the currently running Windows Terminal application.
2. Return to the browser window with the Azure portal.

#### Review

In this exercise, you created a basic function that echoes the content sent via an HTTP POST request.

### Exercise 3: Create a function that triggers on a schedule

#### Task 1: Create a schedule-triggered function

1. Access the **funclogic*[yourname]*** function app that you created earlier in this lab.
2. Create a new function by using the following settings:
   * Development environment: **In-portal**
   * Template: **Timer trigger**
   * Name: **Recurring**
   * Schedule: **0 \* \* \* \* \***

#### Task 2: Observe function runs

1. In the function editor, save the default function implementation.
2. Select **Logs**, and then observe the function run that occurs about every minute. Each function run should render a simple message to the log.

#### Task 3: Update the function integration configuration

1. Back on the **Function Apps** blade, expand the node for the **funclogic*[yourname]*** function app that you created earlier in this lab, and then expand the **Functions** and **Recurring** child nodes.
2. Under the **Recurring** node, select the **Integrate** node.
3. In the **Integrate** section, update the **Timer (my Timer)** configuration with the following values, and then save the configuration changes:
   * Schedule: **\*/30 \* \* \* \* \***

#### Task 4: Observe function runs

1. Back on the **Function Apps** blade, expand the node for the **funclogic*[yourname]*** function app that you created earlier in this lab, and then expand the **Functions** node.
2. Under the **Functions** node, select the **Recurring** node.
3. In the function editor, select **Logs**.
4. Observe the function run that now occurs about every 30 seconds. Each function run should render a simple message to the log.

#### Review

In this exercise, you created a function that runs automatically based on a fixed schedule.

### Exercise 4: Create a function that integrates with other services

#### Task 1: Create an HTTP-triggered function

1. Access the **funclogic*[yourname]*** function app that you created earlier in this lab.
2. Create a new function by using the following settings:
   * Development environment: **In-portal**
   * Template: **HTTP trigger**
   * Name: **GetSettingInfo**
   * Authorization level: **Anonymous**

#### Task 2: Upload sample content

1. Access the **funcstor*[yourname]*** storage account that you created earlier in this lab.
2. Select the **Containers** link in the **Blob service** section, and then create a new container with the following settings:
   * Name: **content**
   * Public access level: **Private (no anonymous access)**
3. Select the recently created **content** container.
4. In the **content** container, select **Upload** to upload the **settings.json** file in the **Allfiles (F): \Allfiles\Labs\02\Starter** folder on your lab VM.

* **Note**: We recommend that you enable the **Overwrite if files already exist** option.

#### Task 3: Configure an HTTP-triggered function

1. Access the **funclogic*[yourname]*** function app that you created earlier in this lab.
2. On the **Function Apps** blade, expand the node for the **funclogic*[yourname]*** function app that you created earlier in this lab, and then expand the **Functions** and **GetSettingInfo** child nodes.
3. Under the **GetSettingInfo** node, select the **Integrate** node.
4. In the **Integrate** section, create a new **input** of type **Azure Blob Storage**.
5. In the **Azure Blob Storage input** section, configure the input with the following details, and then save your configuration changes:
   * Blob parameter name: **json**
   * Path: **content/settings.json**
   * Storage account connection: **Azure WebJobsStorage**
6. Back in the **Integrate** section, select the existing **HTTP trigger**.
7. Update the **HTTP trigger** with the following details, and then save your configuration changes:
   * Allowed HTTP methods: **Selected methods**
   * Request parameter name: **request**
   * Selected HTTP methods: **GET**

#### Task 4: Write function code

1. On the **Function Apps** blade, expand the node for the **funclogic*[yourname]*** function app that you created earlier in this lab, and then expand the **Functions** node.
2. Under the **Functions** node, select the **GetSettingInfo** node.
3. In the function editor, delete the example **run.csx** function script.
4. Add the following **using** directives for libraries that will be referenced by the application:

* using Microsoft.AspNetCore.Mvc;  
  using System.Net;

1. Create a new **public static** method named **Run** that returns a variable of type **IActionResult** and that also takes in variables of type **HttpRequest** and **string** as parameters named *request* and *json*:

* public static IActionResult Run(HttpRequest request, string json)  
  {  
  }

1. Within the **Run** method, return the content of the *json* parameter as the HTTP response of the function:

* return new OkObjectResult(json);

1. Save the updated function script.

#### Task 5: Test function run

1. Open the Windows Terminal application.
2. Start the **httprepl** tool, and then set the base URI to the URL value that you copied earlier in this lab.

* httprepl <function-app-url>
* **Note**: For example, if your URL is **https://funclogicstudent.azurewebsites.net**, your command would be **httprepl https://funclogicstudent.azurewebsites.net**.

1. At the tool prompt, browse to the relative **api/getsettinginfo** endpoint:

* cd api  
  cd getsettinginfo

1. Run the **get** command for the current endpoint:

* get

1. Observe the JSON content of the response from the function app.
2. Exit the **httprepl** application:

* exit

1. Close the currently running Windows Terminal application.
2. Return to the browser window with the Azure portal.

#### Review

In this exercise, you created a function that returns the content of a JSON file in Storage.

### Exercise 5: Clean up your subscription

#### Task 1: Open Azure Cloud Shell and list resource groups

1. In the portal, select the **Cloud Shell** icon to open a new shell instance.
2. If Cloud Shell isn't already configured, configure the shell for Bash by using the default settings.
3. At the **Cloud Shell** command prompt in the portal, enter the following command, and then select Enter to list all resource groups in the subscription:

* az group list

1. Enter the following command, and then select Enter to get a list of possible commands to delete a resource group:

* az group delete --help

#### Task 2: Delete a resource group

1. Enter the following command, and then select Enter to delete the **Serverless** resource group:

* az group delete --name Serverless --no-wait --yes

1. Close the Cloud Shell pane in the portal.

#### Task 3: Close the active application

1. the currently running Microsoft Edge application.

#### Review

In this exercise, you cleaned up your subscription by removing the resource group that was used in this lab.

©2019 Microsoft Corporation. All rights reserved. The text in this document is available under the [Creative Commons Attribution 3.0 License](https://creativecommons.org/licenses/by/3.0/legalcode), additional terms may apply. All other content contained in this document (including, without limitation, trademarks, logos, images, etc.) are ***not*** included within the Creative Commons license grant. This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

This document is provided "as-is." Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it. Some examples are for illustration only and are fictitious. No real association is intended or inferred. Microsoft makes no warranties, express or implied, with respect to the information provided here.