Azure Functions

Lecture 14

Deep Azure@McKesson

Zoran B. Djordjević

Need for serverless computing

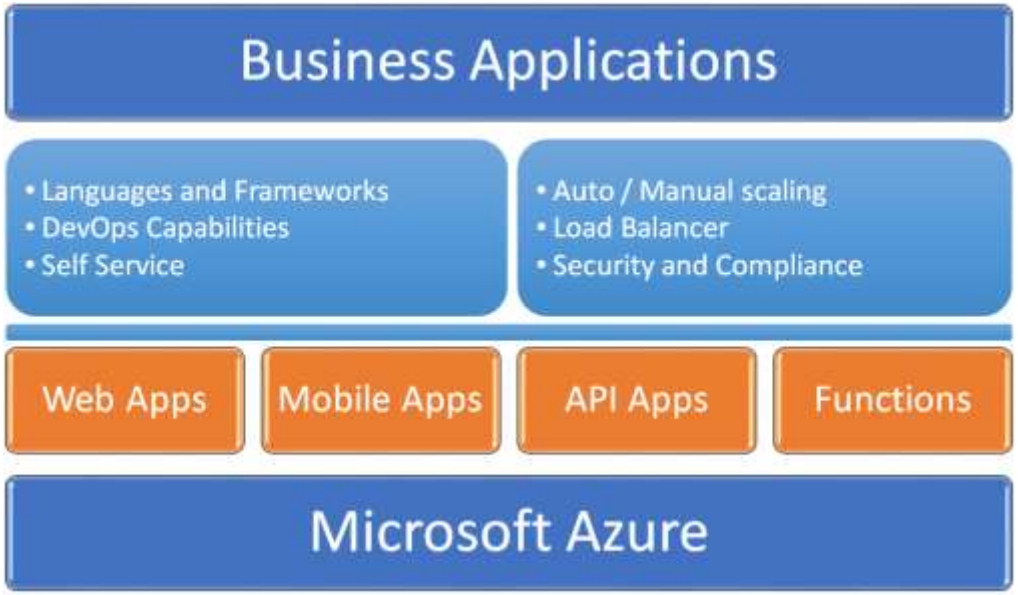
- Cloud has eliminated many if not most of system and database administration tasks and replaced them with highly automated procedures. It is only natural that one looks to an environment that is completely free from any administration.
- Azure Functions and AWS Lambda Functions offer such environments.
- Serverless is not actually serverless. It means that users only need to manage code/application and not servers. The server will be managed by the service provider. We as a user only pay when our code or function is executed in the serverless or in the server that is not managed by us.
- Scaling is based on the request and pricing differs based on the service provider. AWS Lambda and Azure Functions are two examples of serverless computing or **Function as a Service** (**FaaS**).
- AWS provides a pay-as-you-go billing model, while Microsoft Azure provides a consumption plan as well as an App Service plan for Azure Functions.

Benefits

- The following are some of the benefits of serverless computing:
  - Faster time to market as you can write code in the functions editor in the Azure portal and click on **Run** for execution
  - No need to worry about the infrastructure and provisioning resources
  - Easy bindings to services and external services
  - Create functions in multiple languages as supported by the cloud service provider
  - Pay only for what you use
  - More cost-effective than IaaS and PaaS
  - No configuration is required to set up scaling in and scaling out policies

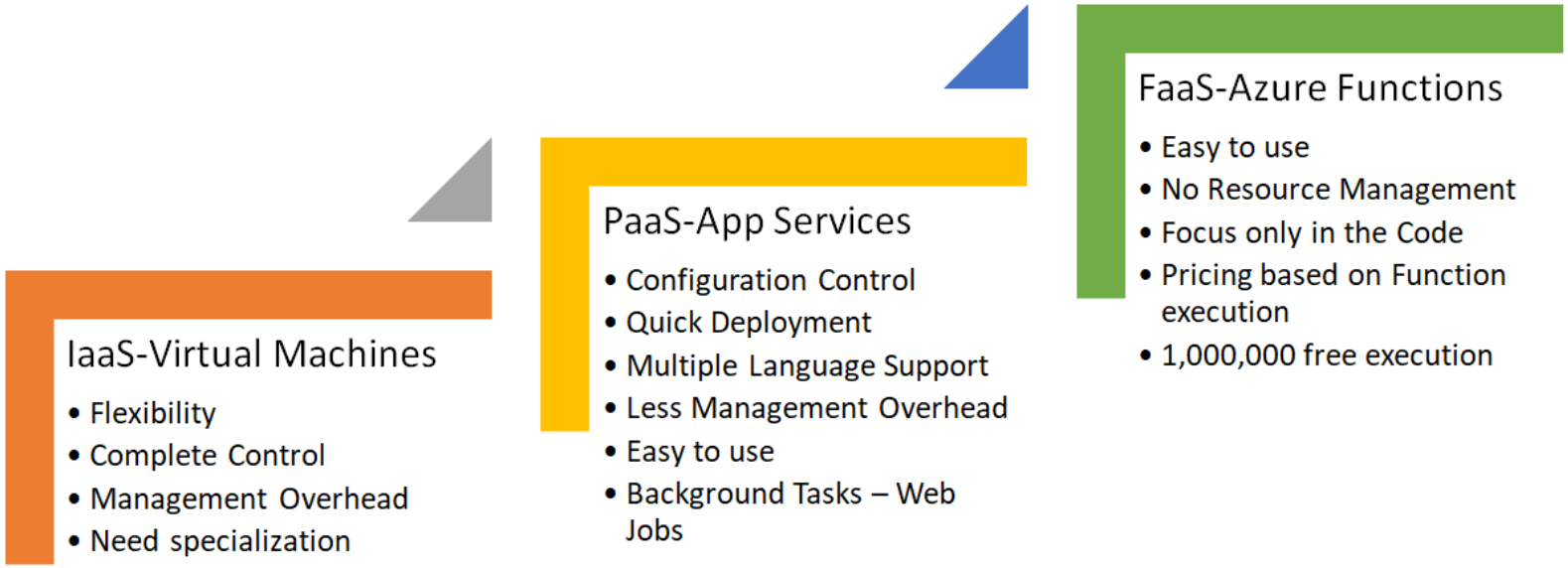
Azure Function

- Azure Function is a service running a function in a cloud environment.
- We use Azure Functions in scenarios where we know the problem, we know the code that can fix the problem, and we don't want to worry about resources that execute this code.
- Azure Function is the easiest way to focus on logic and business and enhance the our productivity.
- We only need to pay for what we use. There are no maintenance of system support costs. There are development costs, though.



Comparison to Other Types of Service

- Azure Functions are similar to Azure Web Jobs with some differences such as scaling policies, trigger events, and language support. We don't need to worry about infrastructure for execution of the piece of code or function.
- We can execute Azure functions in response to events as well.
- The languages that are supported are C#, F#, Node.js, Python or PHP, batch, bash, or PowerShell.



Pricing

There are two types of pricing plans available in the Azure Functions:

- **Consumption plan**: When we execute functions, Microsoft Azure provides all the resources. We only pay for the time that our functions are executed. The consumption plan pricing includes a monthly free grant of 1 million requests and 400,000 GBs of resource consumption per month. Free grants apply to paid consumption subscriptions only.
- **App Service plan**: This executes functions just the way we execute Azure App Services. We can utilize the same App Service plan created for any application and execute Azure Functions on it without any extra cost.

- Having App Service plan as the host for Azure Functions provides lots of benefits that are available with Azure App Services. We can utilize remote debugging, deployment slots, continuous deployment, vertical scaling and horizontal scaling, auto-scaling, and so on.
- If we use the Azure App Service, then it is a multitenant scenario. If we want to utilize a dedicated environment, then we can utilize the App Service Environment that is a dedicated service from Microsoft Azure, where we can host a function in a virtual network and configure **network security groups** (**NSGs**) for an enhanced level of security.

Triggers & Bindings

- Triggers and bindings are the core of Azure Functions.
- Triggers allow us to write a function to respond to events that occur in the Azure or other services. As the name suggests, trigger indicates how the function should be invoked. AzureFunctions can have only one trigger associated with it
- Bindings specify how is a particular function related to data. Bindings are the connection to the data from within the code available in the function. Unlike triggers, functions can have one or more input and output bindings.
- The table on the following slide shows which triggers and bindings that are supported with Azure Functions.

- All triggers have associated input data.
- The HTTP output binding requires an HTTP trigger.

Triggers & Bindings

Type	Service	Trigger	Input	Output
Schedule	Azure Function	Yes		
HTTP	Azure Functions	Yes		Yes**
Blob Storage	Azure Storage	Yes	Yes	Yes
Events	Azure Event Hubs	Yes		Yes
Queues	Azure Storage	Yes		Yes
Queues and Topics	Azure Service Bus	Yes		Yes
Storage tables	Azure Storage		Yes	Yes
SQL tables	Azure Mobile Apps		Yes	Yes
No-SQL DB	Azure DocumentDB		Yes	Yes
Push Notifications	Azure Notification Hubs			Yes
Twilio SMS Text	Twilio			Yes
SendGrid email	SendGrid			Yes

App Service Plan

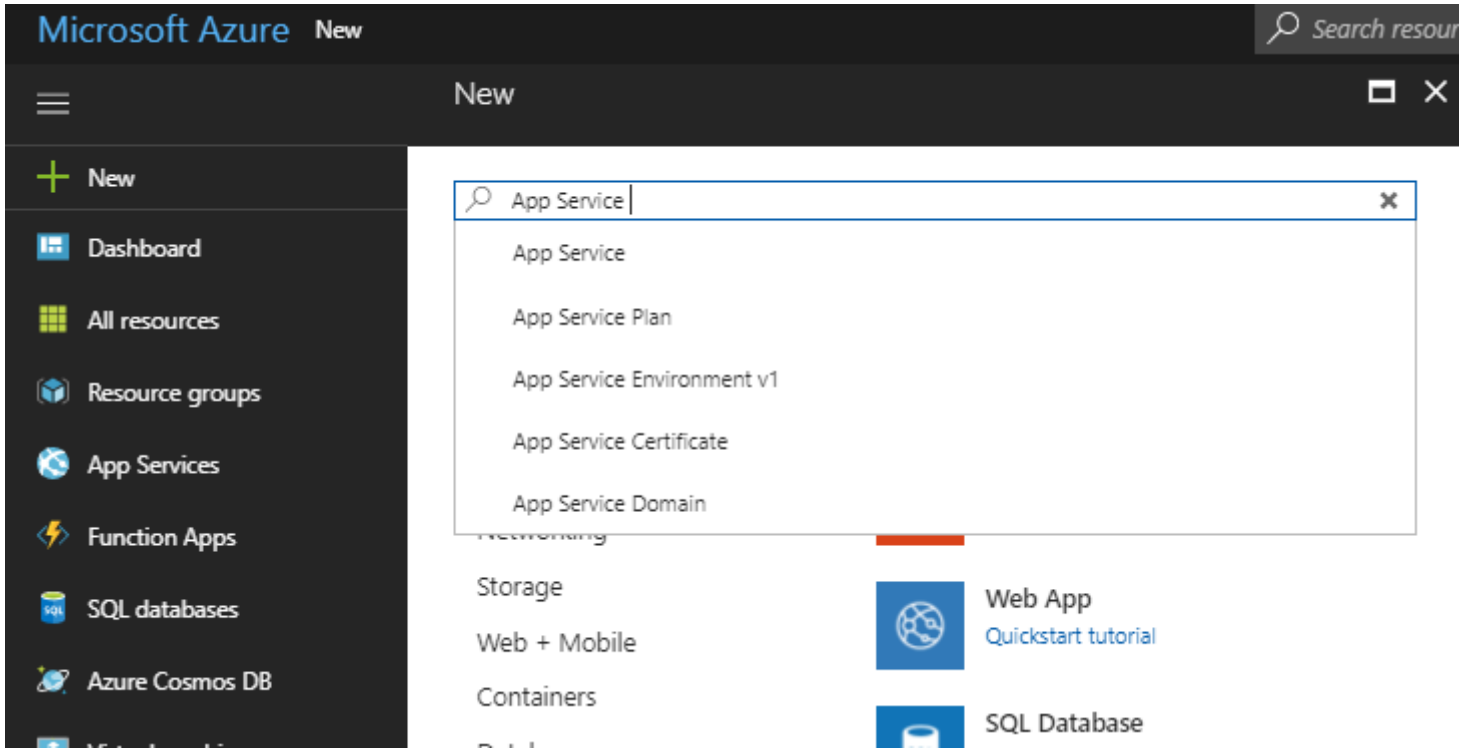
- An **App Service plan** (**ASP**) is a combination of capacities (instance size and instance count on which the application is hosted) and features.
- Capacity is directly related to cost and hence it is similar to choosing a pricing tier. There are different capabilities and limits available in pricing plans.
- Each ASP can be used for different purposes and they provide different features too.
- There are five pricing tiers as follows:
  - **Free**: no scaling
  - **Shared**: no scaling
  - **Basic**: SLA - 99.95%; maximum instances for scaling - 3
  - **Standard**: SLA - 99.95%; autoscale, 5 deployment slots; Geo-distributed deployment, VPN hybrid connectivity, deployment slots, and automated backups; maximum instances for scaling - 10
  - **Premium**: SLA - 99.95%; 20 deployment slots; autoscale, geo-distributed deployment, VPN hybrid connectivity, deployment slots, and automated backups; maximum instances for scaling - 50

Features of App Service Plan

- An App Service plan can be shared by multiple applications.
- Deployment slots are usually deployed on the same App Service plan.
- Azure Web Apps configured with an App Service plan are changed, and then these changes affect all the applications hosted on the App Service plan.
- By default, ASP comes with a single instance. If we increase the instance count, then the applications hosted on a single instance will be hosted on other instances too.
- The number of instances in ASP is directly associated with the price of Azure Web Apps

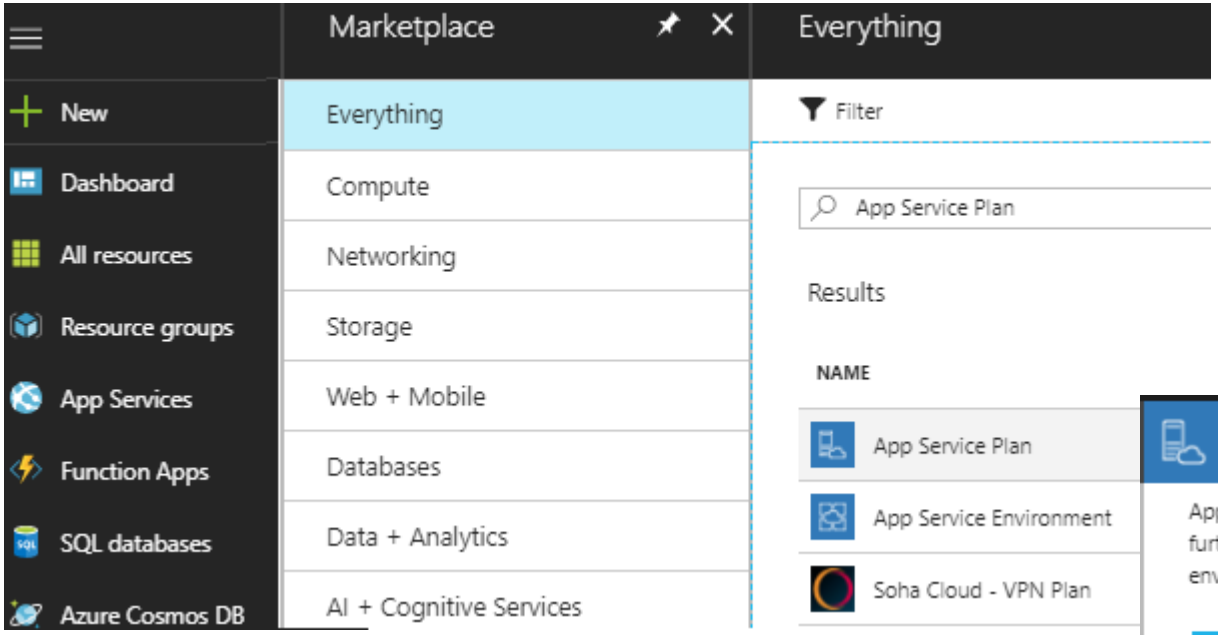
Create App Service Plan

- To create an App service plan, open Azure portal, hit + New and type App Service plan in the search field of New pane that pops-up.

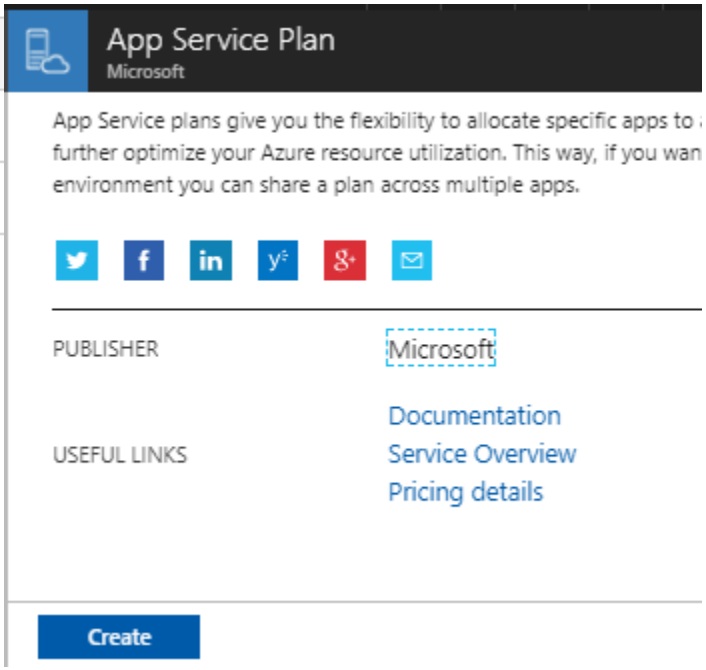


Select App Service Plan

- On the next (Everything) pane that shows up, select App Service Plan



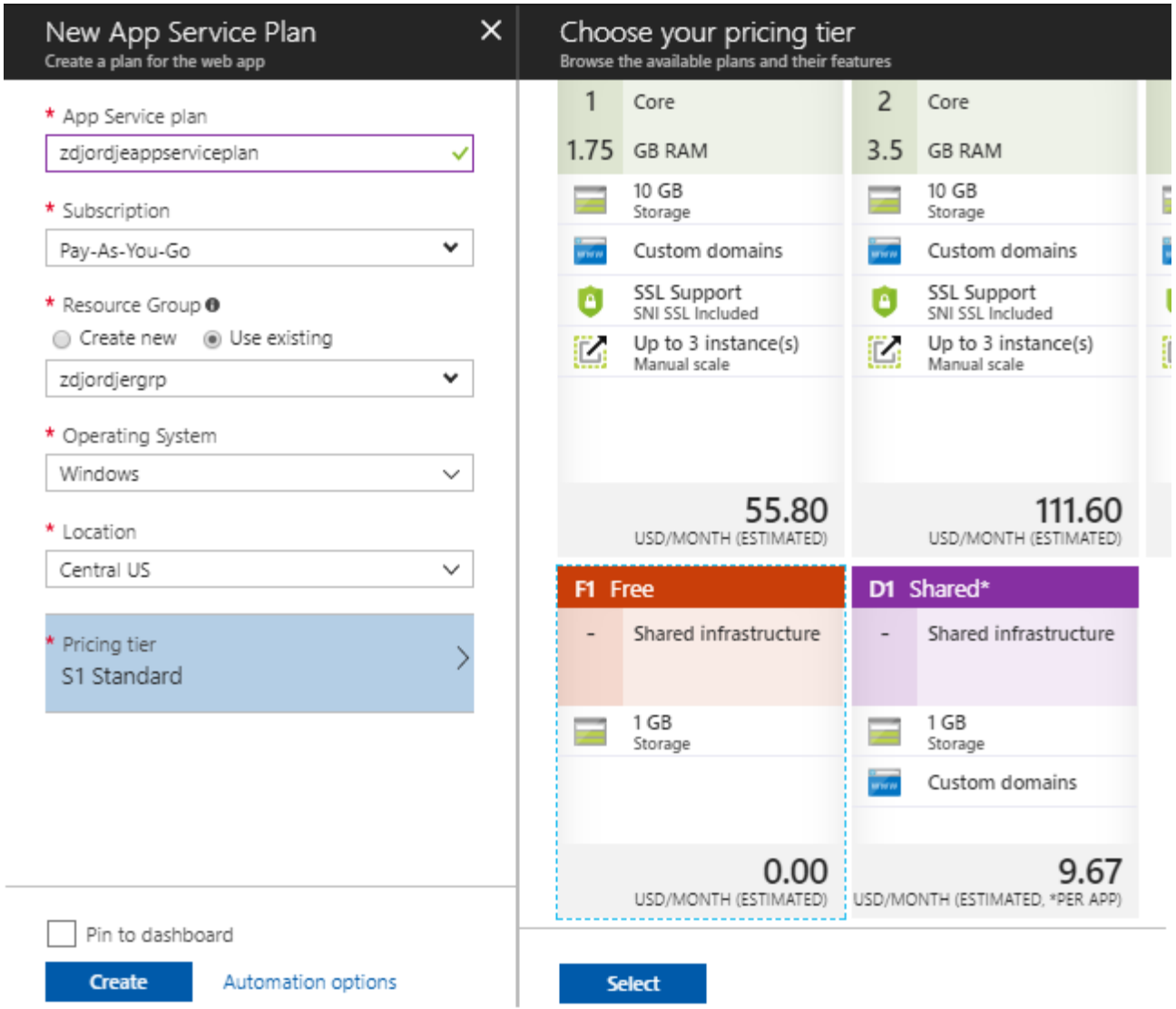
- On the following pane hit Create



Parameters of App Service Plan

- Populate details, name, resource group, location, Pricing tier. Hit Select and Create

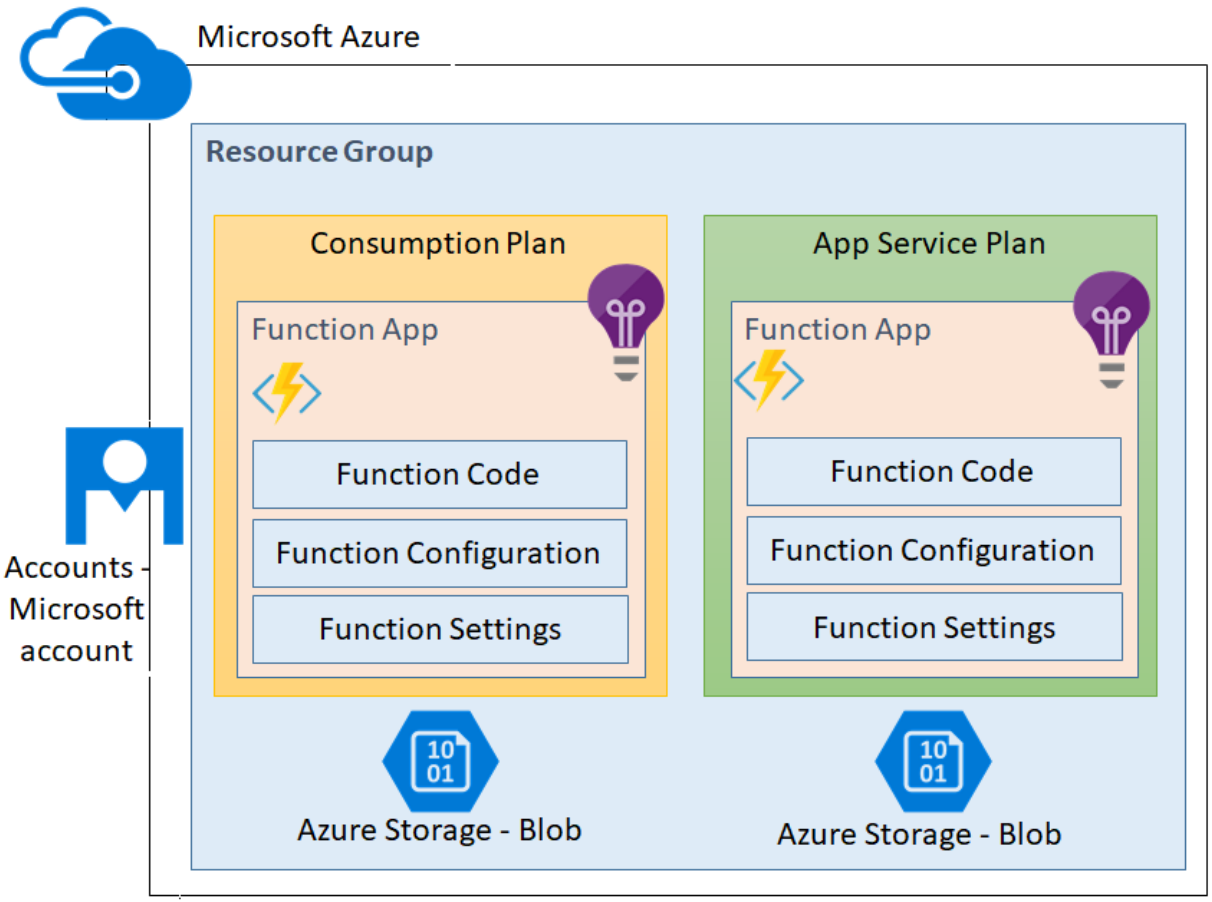
- After a short while, App service Plan will be deployed and will be visible in my resource group.



Create an Azure Function

- We will create an Azure Function App and then we will create an Azure Function with triggers and output bindings.

- One of the most important things in Azure App Services or Azure Functions is the Kudu editor which we could use to troubleshoot any issues in our functions
- We will go through in detail about the Kudu editor, folder structure, and files available in the Azure Functions.
- Azure function can exist in two environments: App Service Plan and Consumption Plan.



Azure Function App, Function Code

- A function app is a collection of one or more functions that are managed together. All the functions in a Function App share the same pricing plan and it can be a consumption plan or an App Service plan.
- When we utilize Visual Studio Team Services for Continuous Integration and Continuous Delivery using build and release definitions, then the Function app is also shared.
- The way we manage different resources in Azure with the Azure Resource Group is similar to how we can manage multiple functions with the Function App.

- In the following, we will consider a scenario where photographers need to upload photographs to the portal. The moment a photograph is uploaded, a thumbnail should be created immediately.
- As an illustration we will use a JavaScript code, presented on the following slide.

Function Code

```
var Jimp = require("jimp");
// JavaScript function must export a single function via module.exports // To find the function and execute it
module.exports = (context, myBlob) => {
// context is a must have parameter and first parameter always, // context is used to pass data to and from the function
//context name is not fixed; it can be anything
	// Read Photograph with Jimp
	Jimp.read(myBlob).then((image) => {
	// Manipulate Photograph,  resize the Photograph. Jimp.AUTO can be passed as one of the values.
	image
		.resize(200, 200).quality(40).getBuffer(Jimp.MIME_JPEG, (error, stream) => {
			// Check for errors while processing the Photograph.
			if (error) {
				// To print the message on log console
				context.log('There was an error processing the Photograph.');
				// To communicate with the runtime that function is finished to avoid timeout
				context.done(error);
			}
			else {
				// To print the message on log console
				context.log('Successfully processed the Photograph');
				// To communicate with the runtime that function is finished to avoid timeout
				// Bind the stream to the output binding to create a new blob
				context.done(null, stream);
			}
		});
	});
};
```

Function Configuration

- Function configuration defines the function bindings and other configuration settings. It contains configurations such as the type of trigger, paths for blob containers, and so on:

```
{
    "bindings": [
        {
            "name": "myBlob",
            "type": "blobTrigger",
            "direction": "in",
            "path": "photographs/{name}",
            "connection": "origphotography2018_STORAGE",
            "dataType": "binary"
        },
        {
            "type": "blob",
            "name": "$return",
            "path": "thumbnails/{name}",
            "connection": "origphotography2018_STORAGE",
            "direction": "out"
        }
    ],
    "disabled": false
}
```

- The function runtime uses this configuration file to decide which events to monitor and how to pass data to and from the function execution.

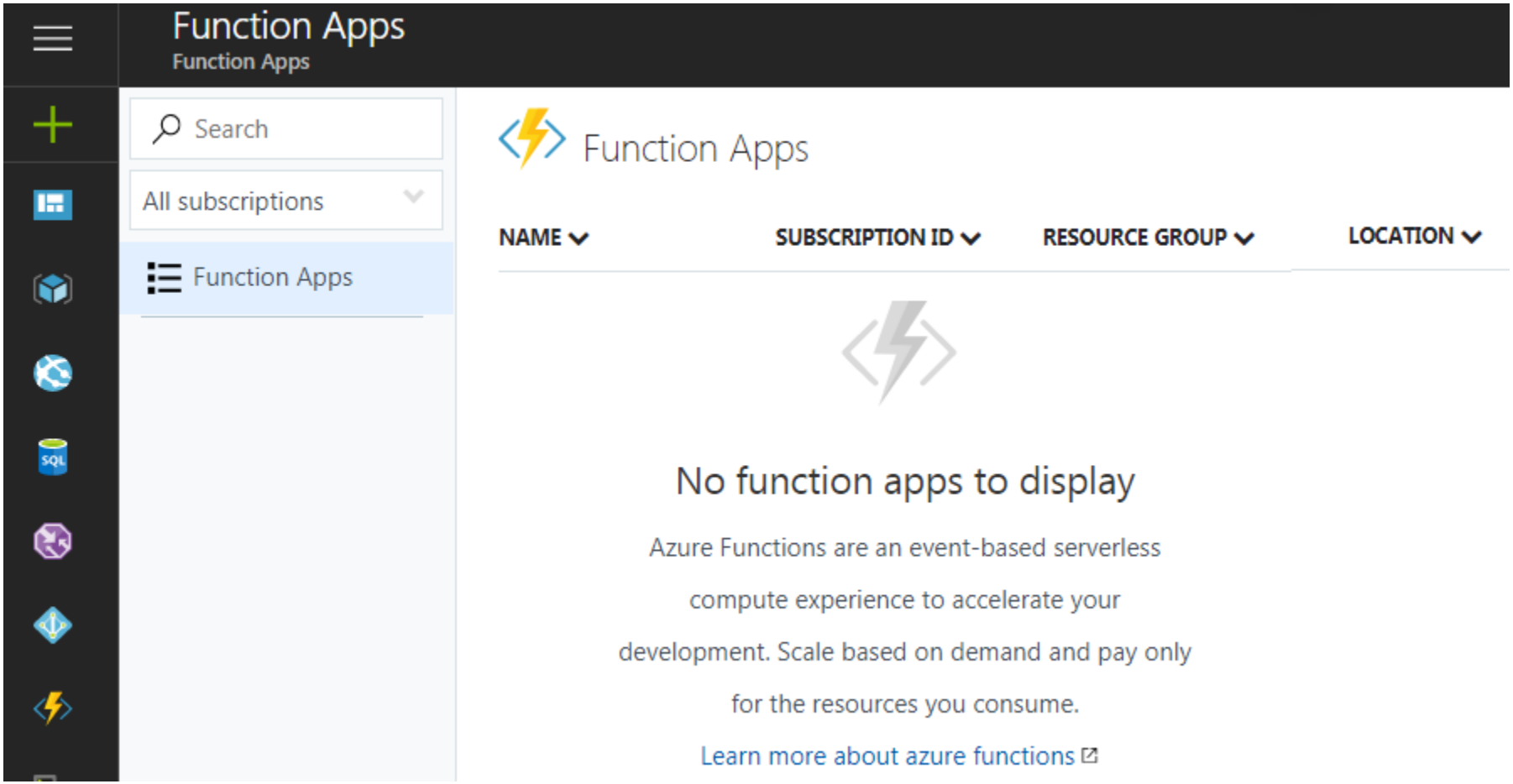
Function Settings & Runtime

- We can limit the daily usage quota and application settings. We can enable Azure Function proxies and change the edit mode of our function app.
- The application settings in the Function App are similar to the application settings in Azure App Services.
- We can configure .NET Framework v4.6, Java version, Platform, ARR Affinity, remote debugging, remote Visual Studio version, app settings, and connection strings.

- The runtime is responsible for executing function code on the underlying WebJobs SDK host.

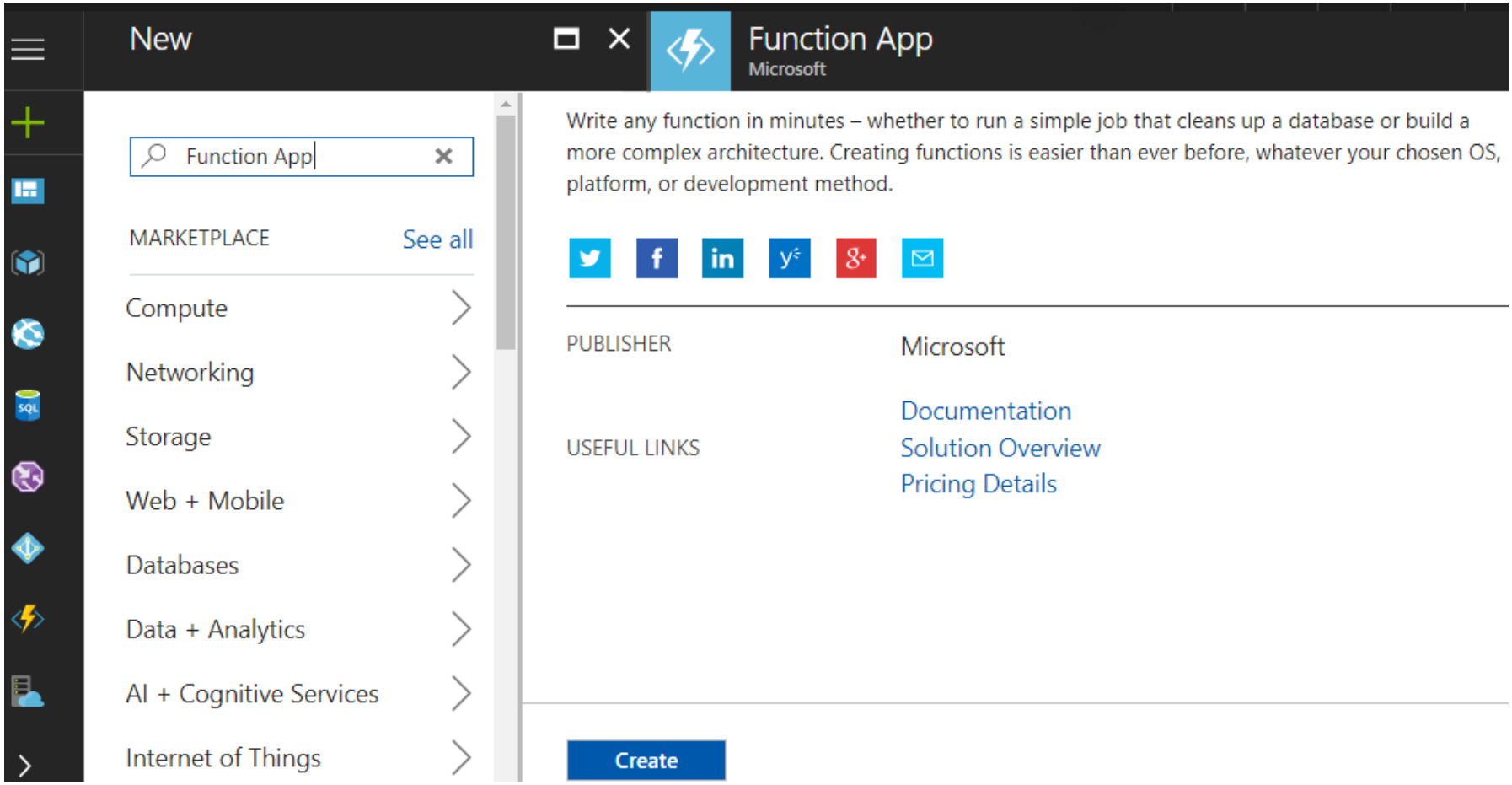
Setting up an Azure Function

- In Azure portal we select + New, Function Apps. Initially we have no function apps.



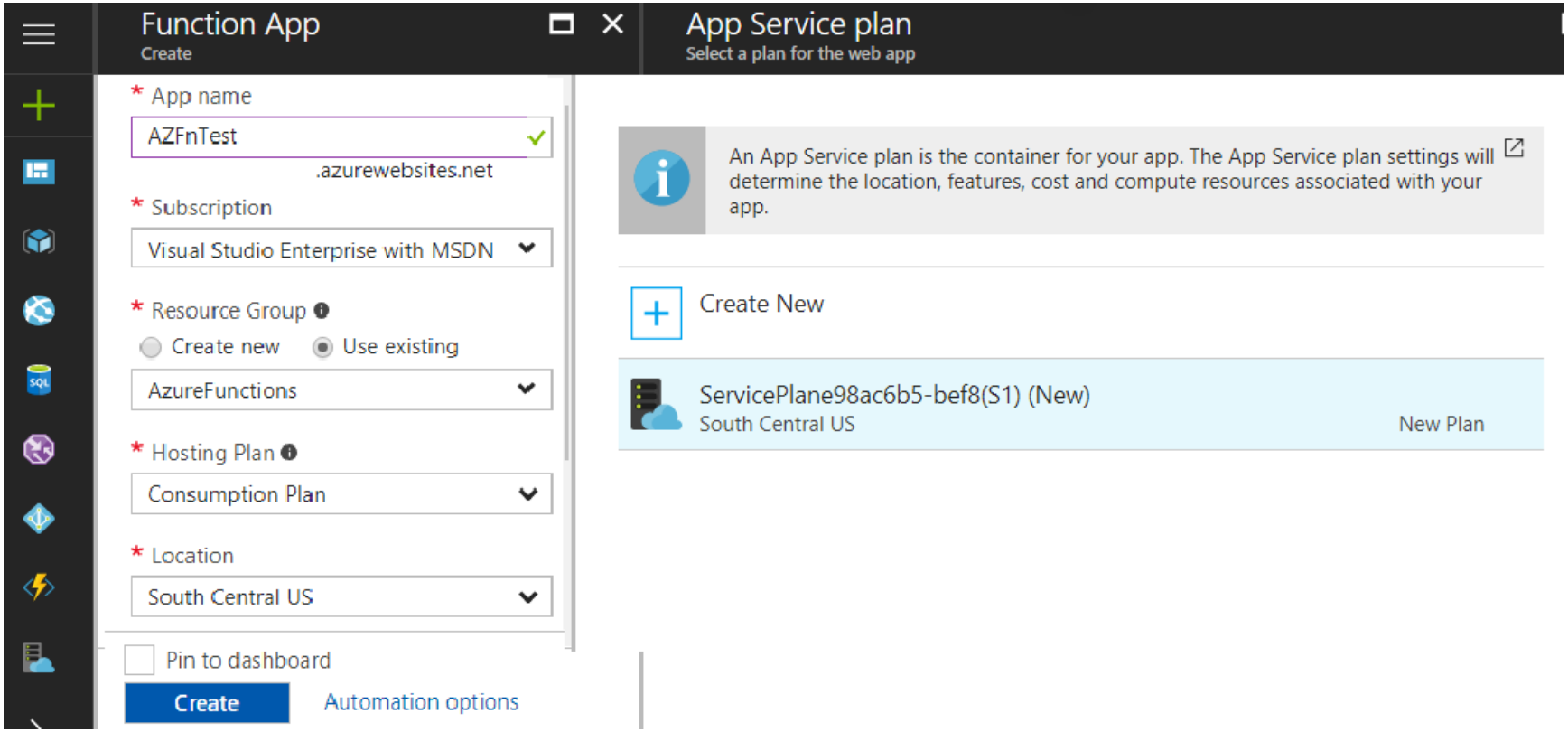
Create Function App

- On Function App pane, select Create



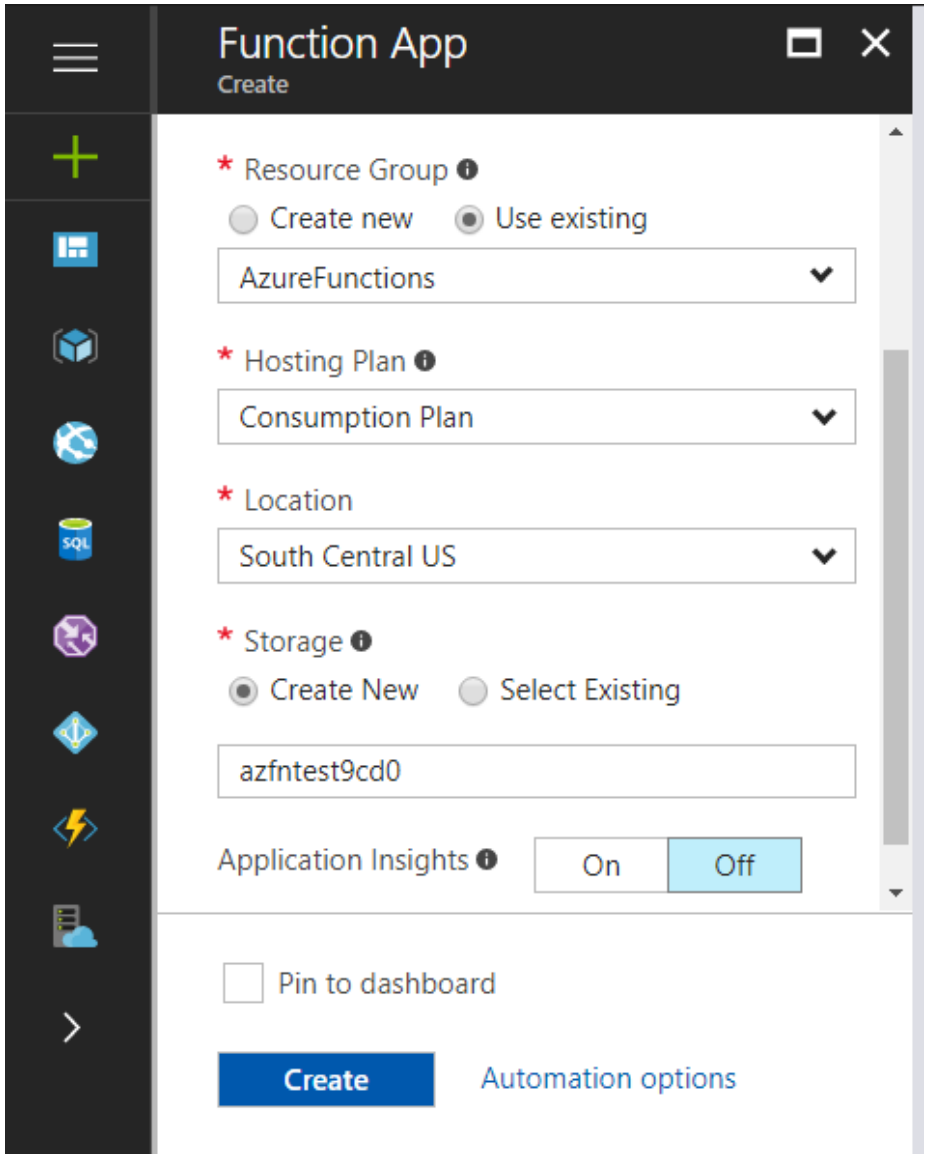
Provide App name, other details

- Provide the **App name**, **Subscription** details, and existing **Resource Group**.
- Select **Consumption Plan or App Service Plan** in **Hosting Plan**. Then select **Location**:



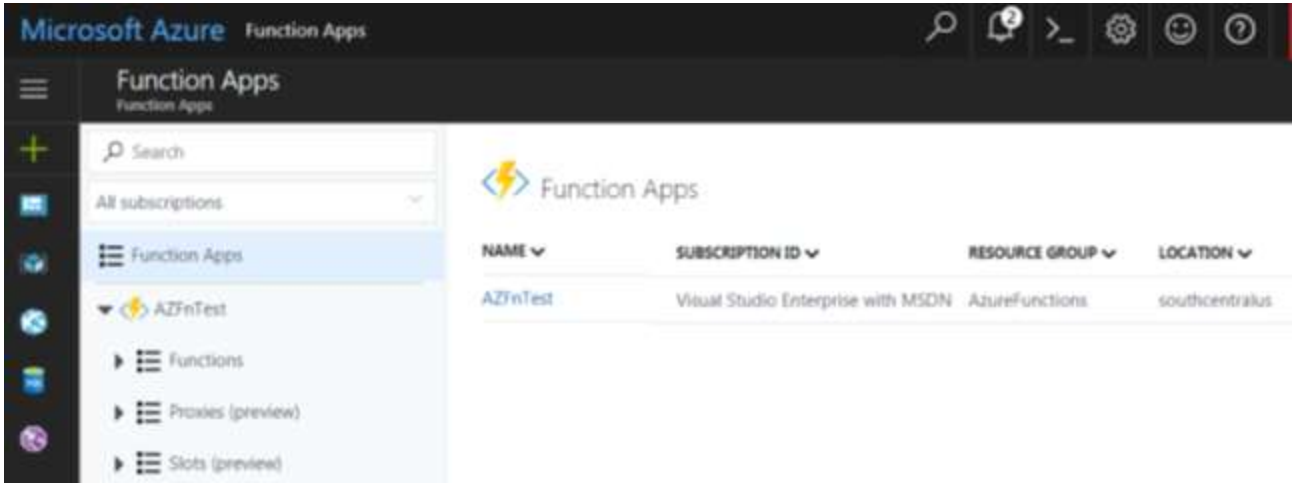
Create New Storage

- Select **Create New** in **Storage** and click on **Create**:

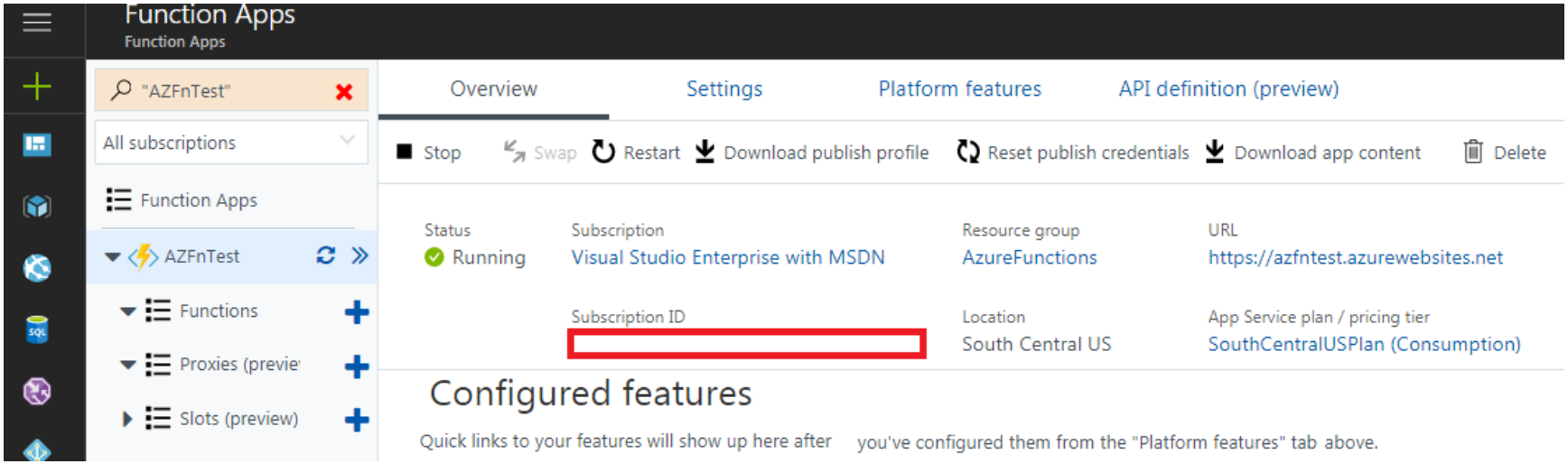


Verify Function App Exist

- Now, let's go to **Function Apps** in the left sidebar and verify whether the recently created Function App is available in the list or not:



- Click on the Function App and we see the details: Subscription, Resource group, URL, Location, App Service Plan / pricing tier. We can Stop/Start function app



Settings

- The **Settings** tab provides details on the **Runtime version**, **Application settings**, and the limit on daily usage:

Function Apps
Function Apps

"AZFnTest"

All subscriptions

Function Apps

AZFnTest

Functions

Proxies (preview)

Slots (preview)

Overview | Settings | Platform features | API definition (preview)

Daily Usage Quota (GB-Sec)

1

Set quota

Application settings

Manage application settings

Runtime version

Runtime version: 1.0.11002.0 (~1)

Proxies (preview)

Enable Azure Functions Proxies (preview)

Off | On

Function app edit mode

Change the edit mode of your function app

Read\Write | Read Only

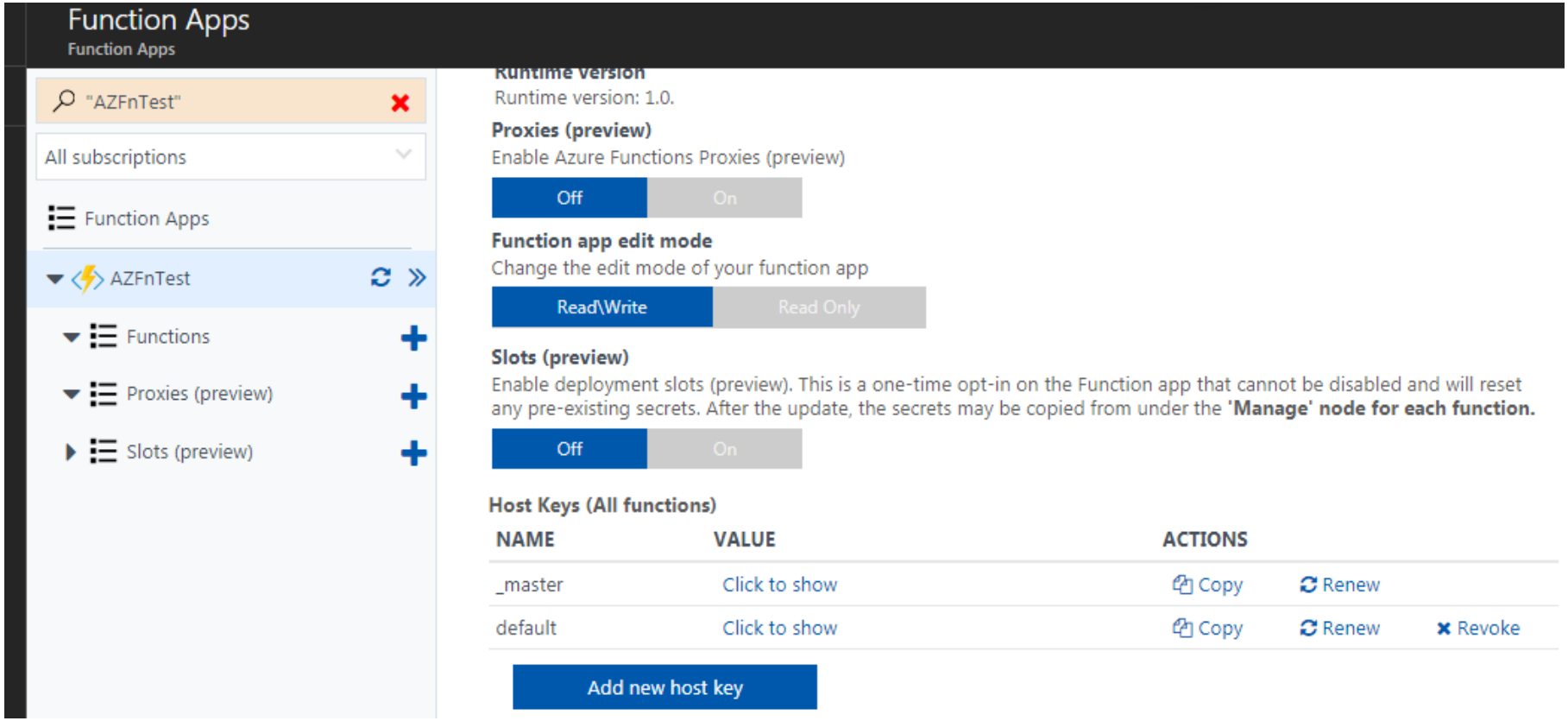
Slots (preview)

Enable deployment slots (preview). This is a one-time opt-in on the Function app that cannot be disabled and will reset any pre-existing secrets. After the update, the secrets may be copied from under the **'Manage' node for each function.**

Off | On

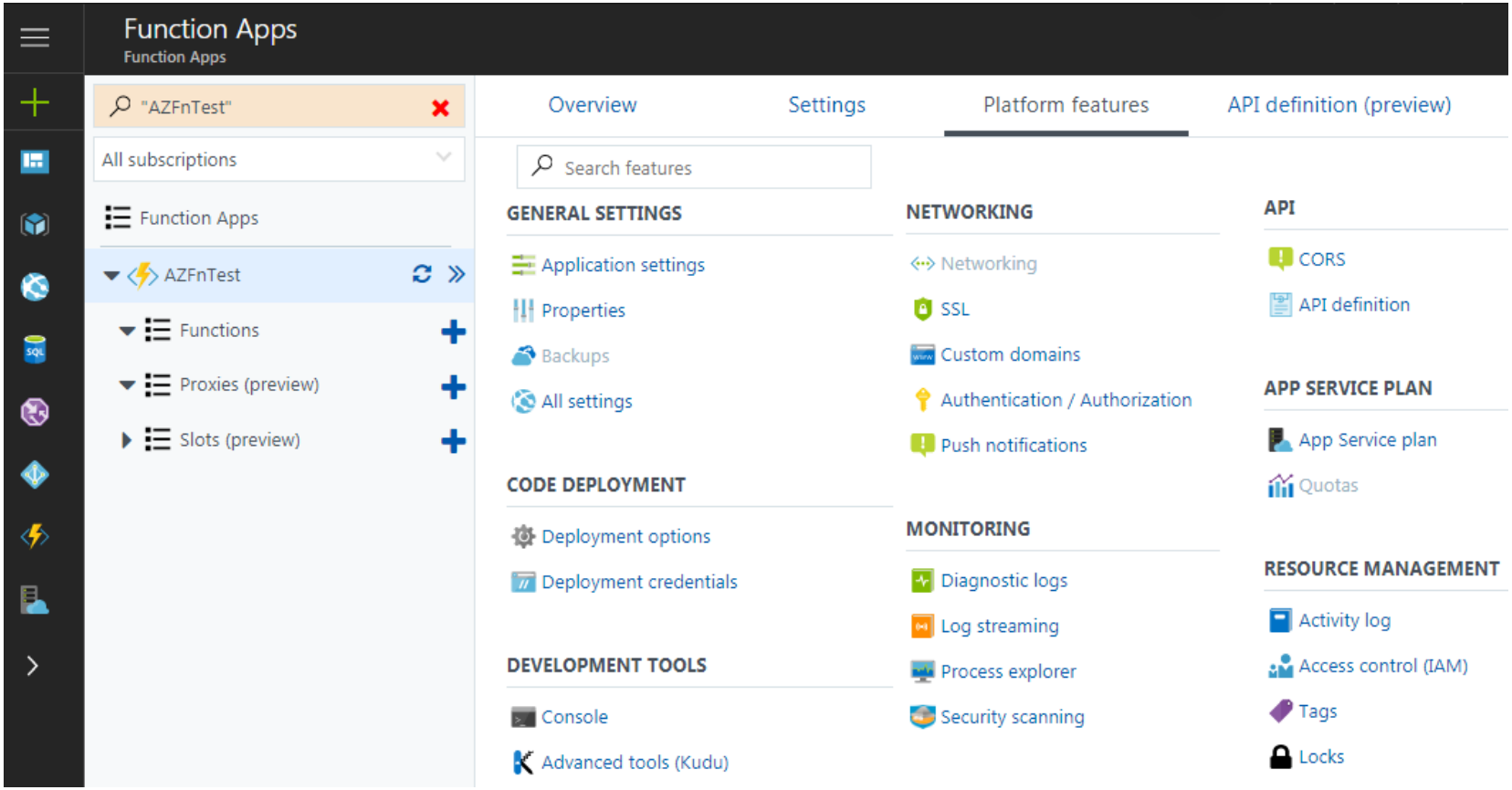
Settings

- Settings also allows us to keep the Function App in **Read/Write** or **Read Only** mode. We can also enable deployment slots, a well-known feature of Azure App Services:



Platform Features Tab

- In the **Platform features** tab as shown below, we get different kinds of options to enable the Function App with **MONITORING**, **NETWORKING**, **DEPLOYMENT TOOLS**, and so on.



Properties

- Click on **Properties**. Verify the different details that are available:

Microsoft Azure Function Apps > Properties

Overview Settings Platform features API definition

Search features

GENERAL SETTINGS
- Application settings
- Properties
- Backups
- All settings

CODE DEPLOYMENT
- Deployment options
- Deployment credentials

DEVELOPMENT TOOLS
- Console
- Advanced tools (Kudu)

NETWORKING
- Networking
- SSL
- Custom domains
- Authentication / Authorization
- Push notifications

MONITORING
- Diagnostic logs
- Log streaming
- Process explorer
- Security scanning

API
- CORS
- API definition

APP SERVICE PLAN
- App Service plan
- Quotas

RESOURCE MANAGEMENT
- Activity log
- Access control (IAM)
- Tags
- Locks

Properties
azfntest

STATUS
Running

URL
azfntest.azurewebsites.net

VIRTUAL IP ADDRESS
No IP-based SSL binding is configured

MODE
Consumption

OUTBOUND IP ADDRESSES
[redacted].89.24,104.215.89...

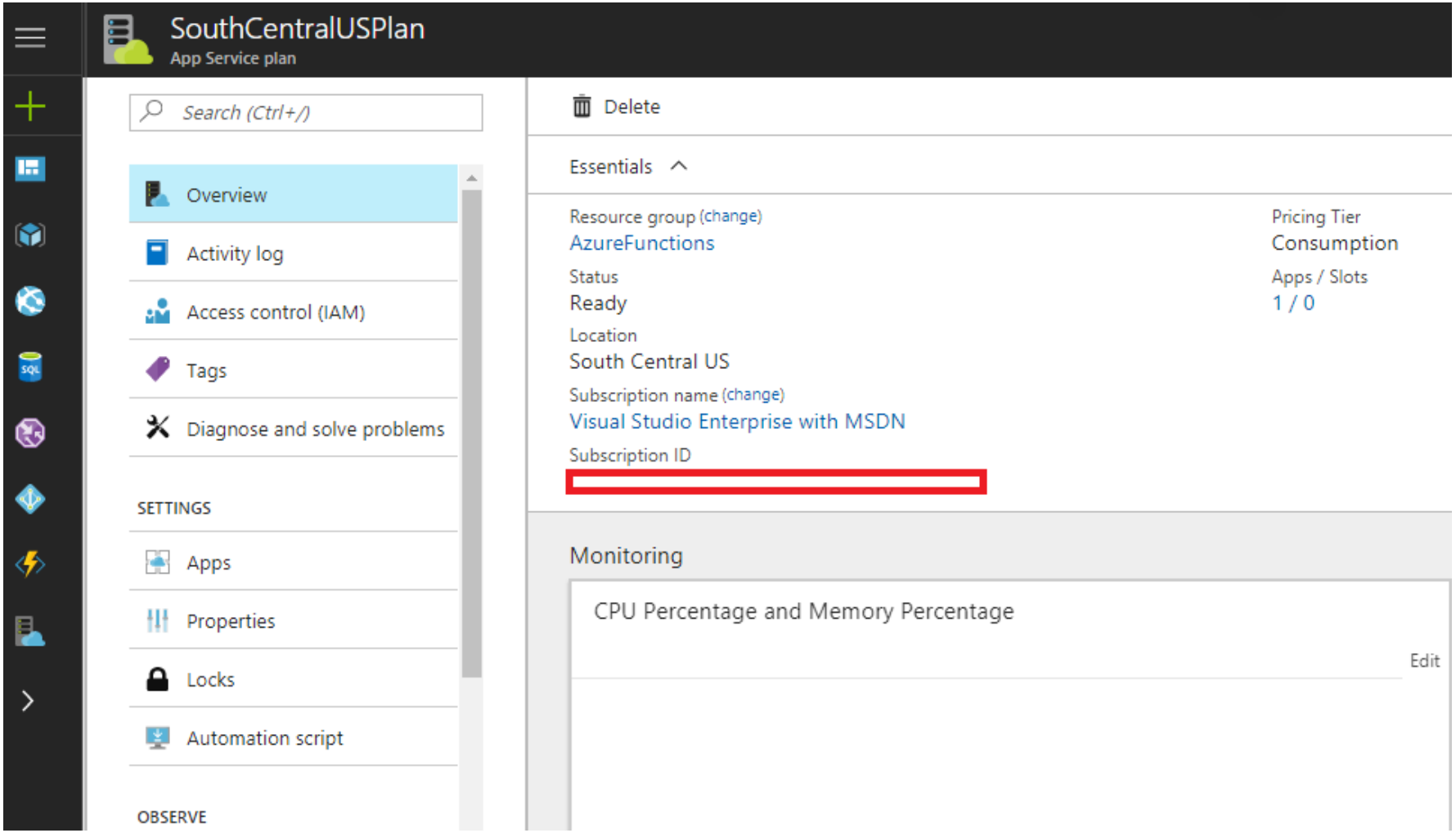
DEPLOYMENT TRIGGER URL
https://$AZFnTest:2PnT0ibKu7YtrQi1gc...

FTP/DEPLOYMENT USER

- There is a property named **OUTBOUND IP ADDRESSES** which is useful if we need the IP addresses of the Function App for whitelisting.

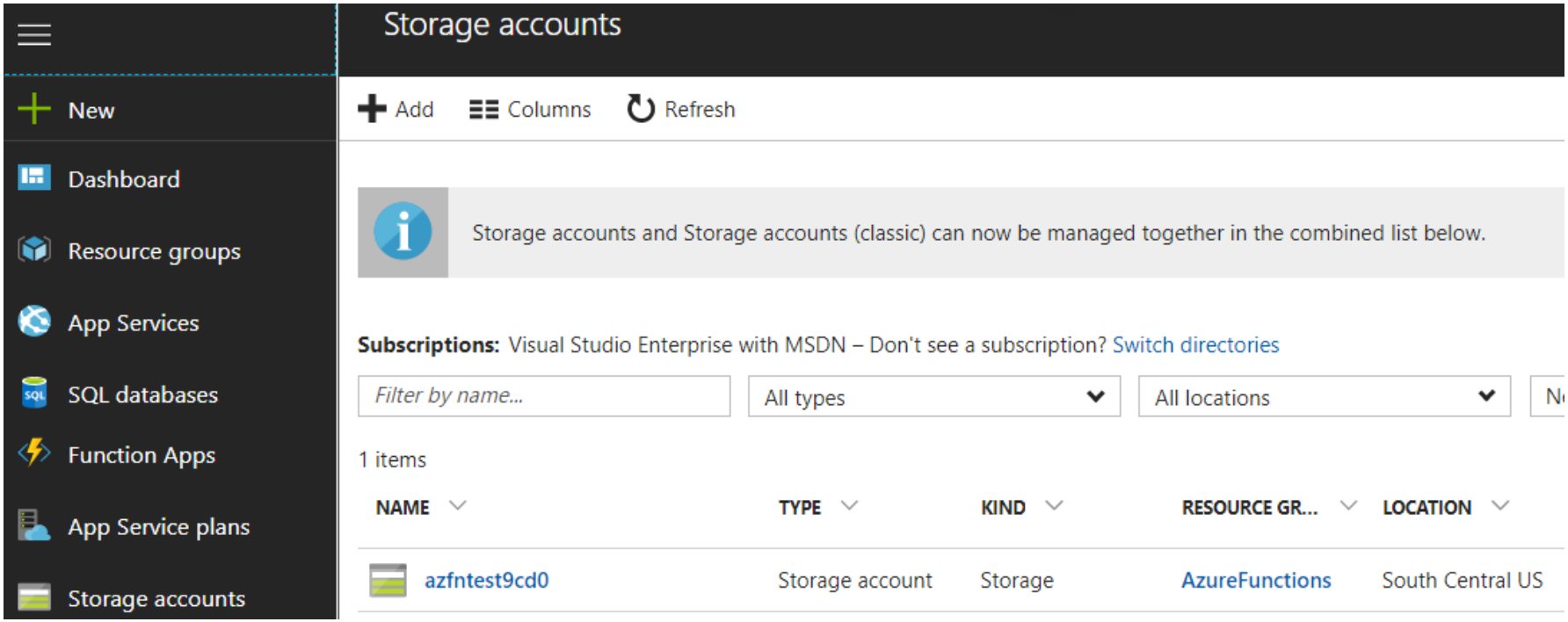
App Service Plan

- Click on **App Service plan** and it will open a consumption plan in the pane:



Storage

- On the left sidebar in the Azure Portal, go to **Storage** services and verify the storage accounts that are available:



- What we want to achieve is that when we upload an image in a specific blob container, the function should be available immediately in the Function App and should be executed and create a thumbnail in another blob container.

Create 2nd Storage Account

Storage accounts
(Default Directory)

Add | Columns | Refresh

Storage accounts and Storage accounts (classic) can now be managed together in the combined list below.

Subscriptions: Visual Studio Enterprise with MSDN – Don't see a subscription? Switch directories

Filter by name...

1 items

NAME

azfntest9cd0 •••

Create storage account

usage and the options you choose below.
Learn more

* Name
origphotography2018 ✓
.core.windows.net

Deployment model
Resource manager | Classic

Account kind
General purpose

Performance
Standard | Premium

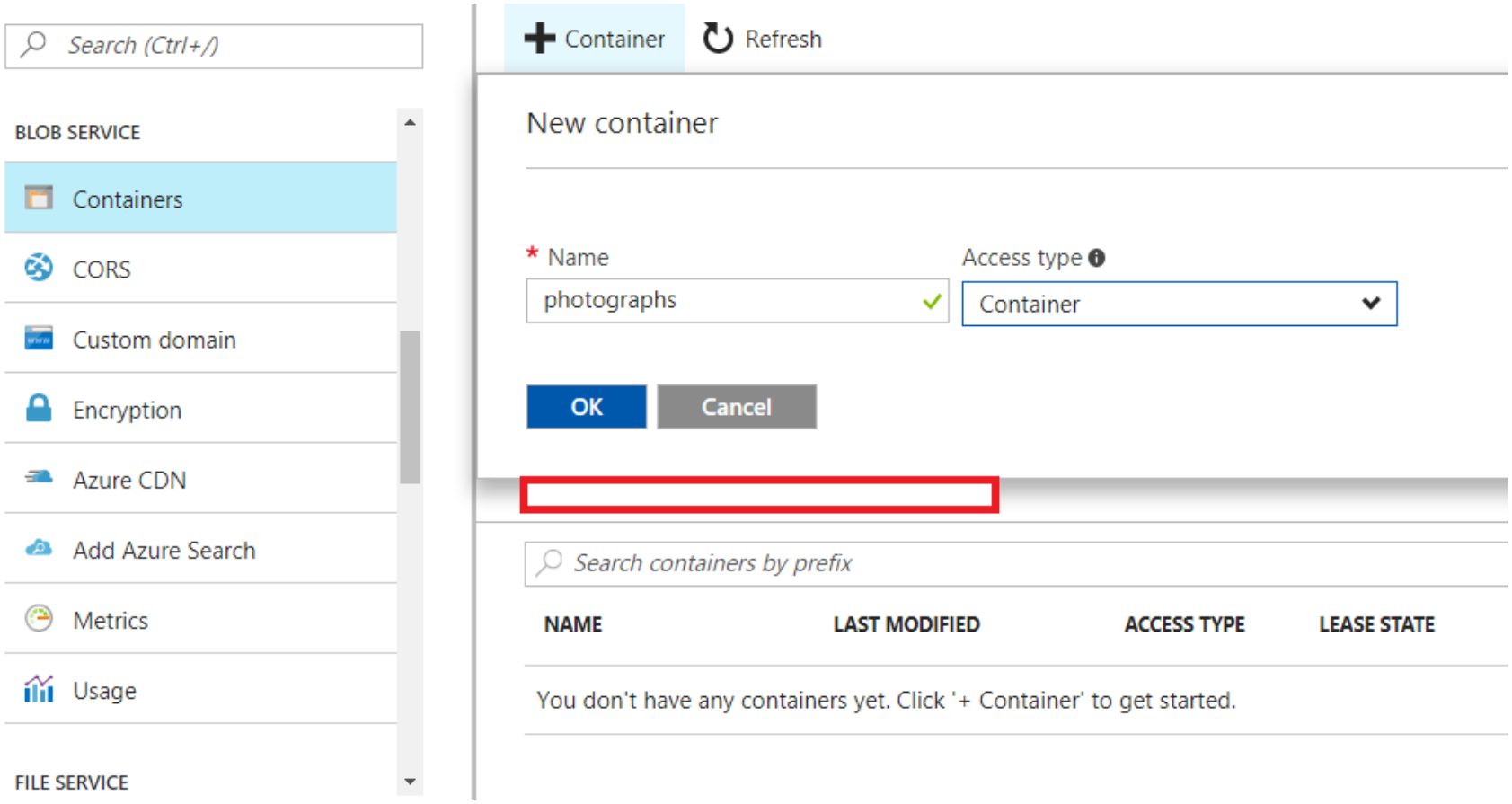
Replication
Read-access geo-redundant storage (RA...

Pin to dashboard

Create | Automation options

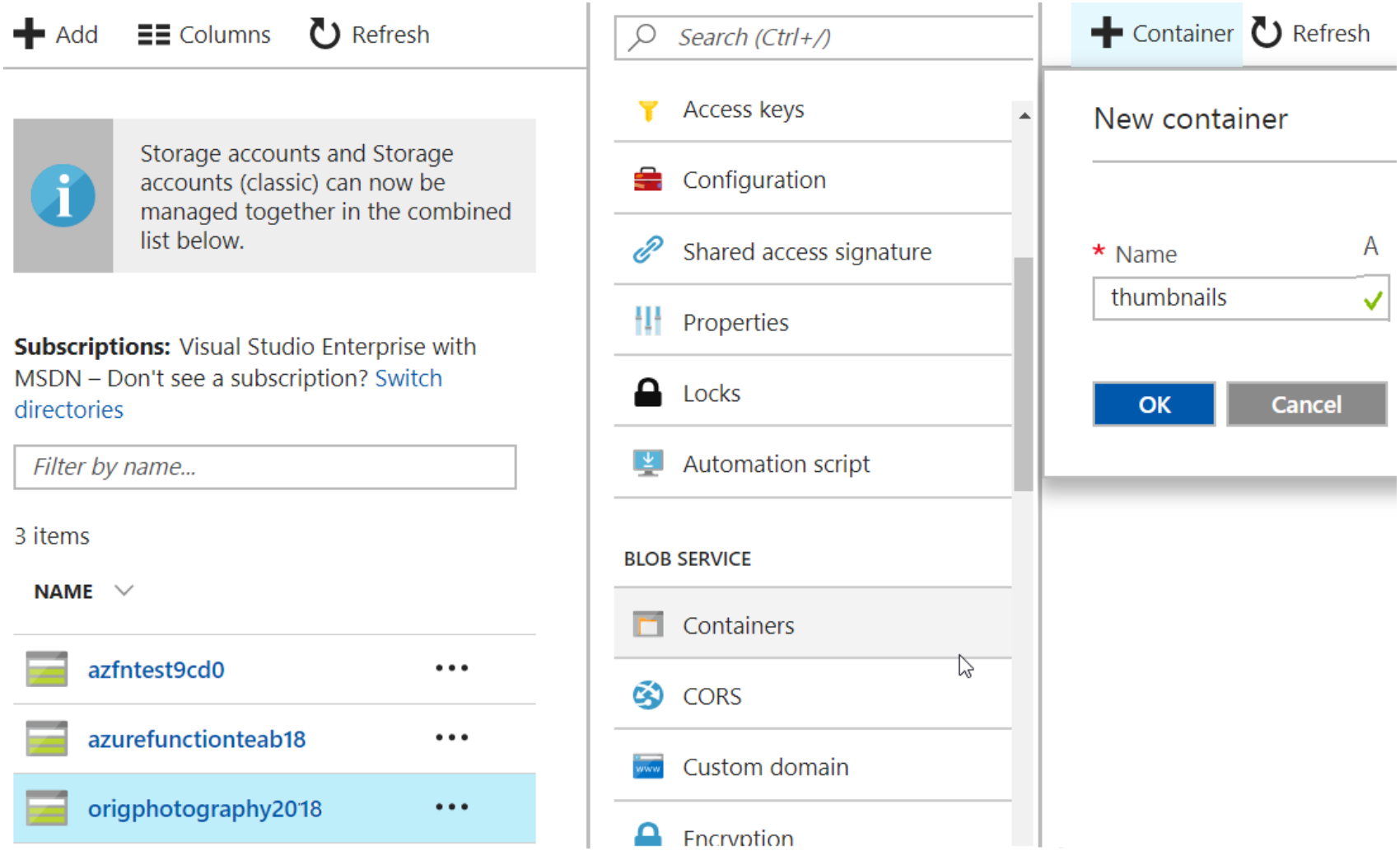
Create Container for Photos

- We need containers in both storage accounts. Click on **+ Container** and fill in the **Name** and **Access type** and click on **OK**:

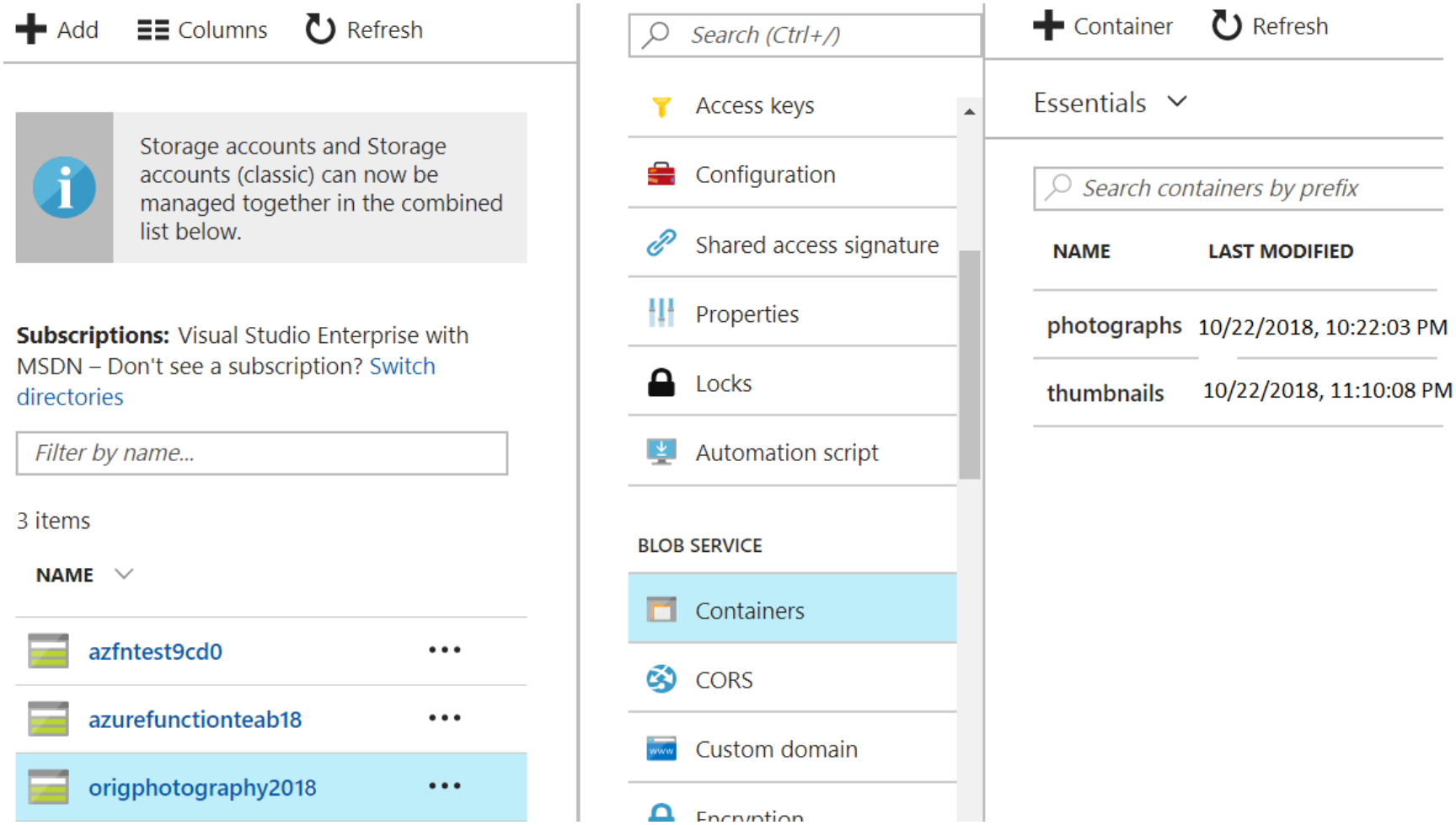


Create Containers for Thumbnails

- Similarly create a container for thumbnails



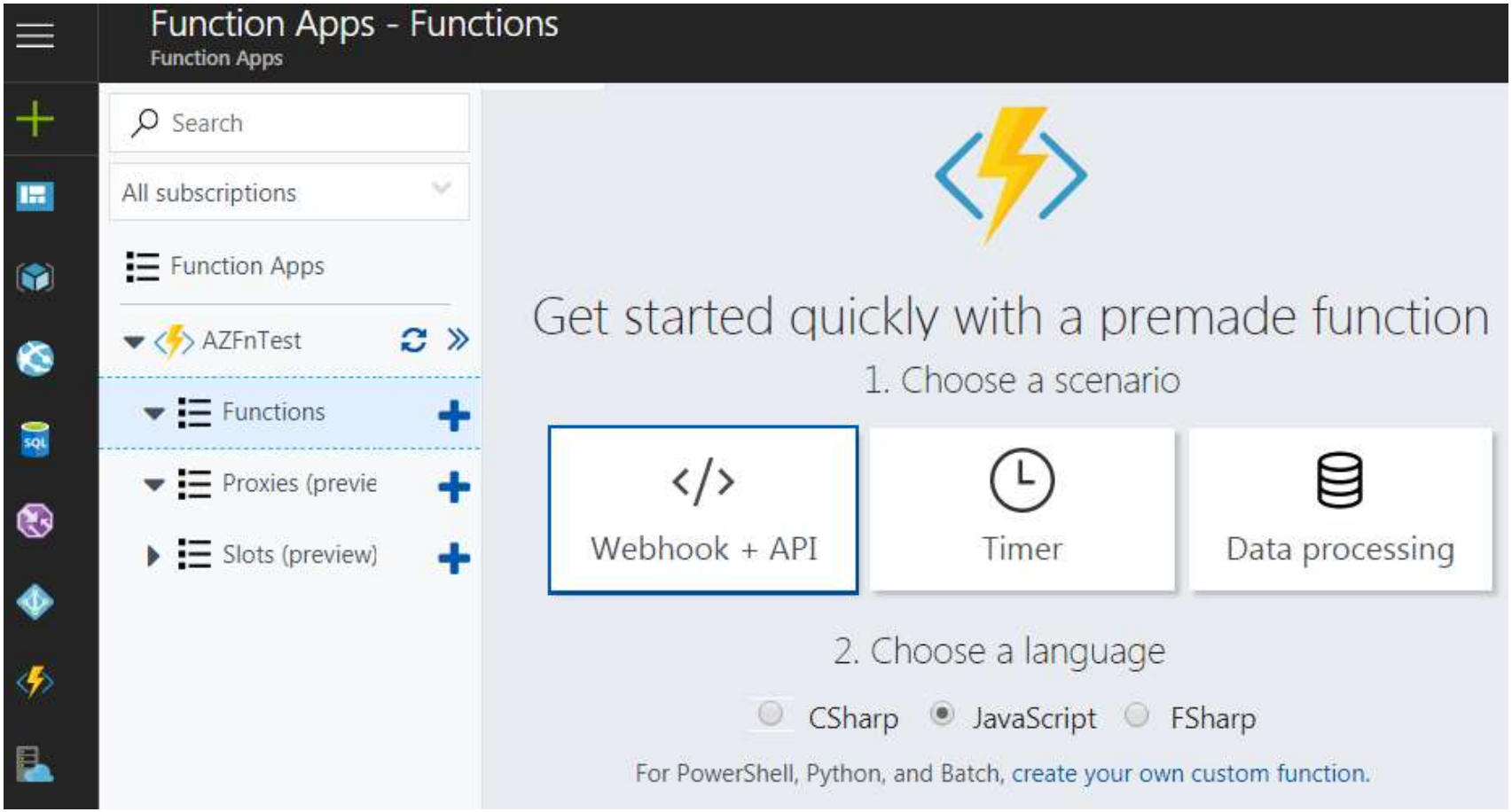
Verify Containers in the Storage Accounts



- We have all the components ready to achieve our main objective of creating a function that creates thumbnails of photographs, and we can start creating a function.

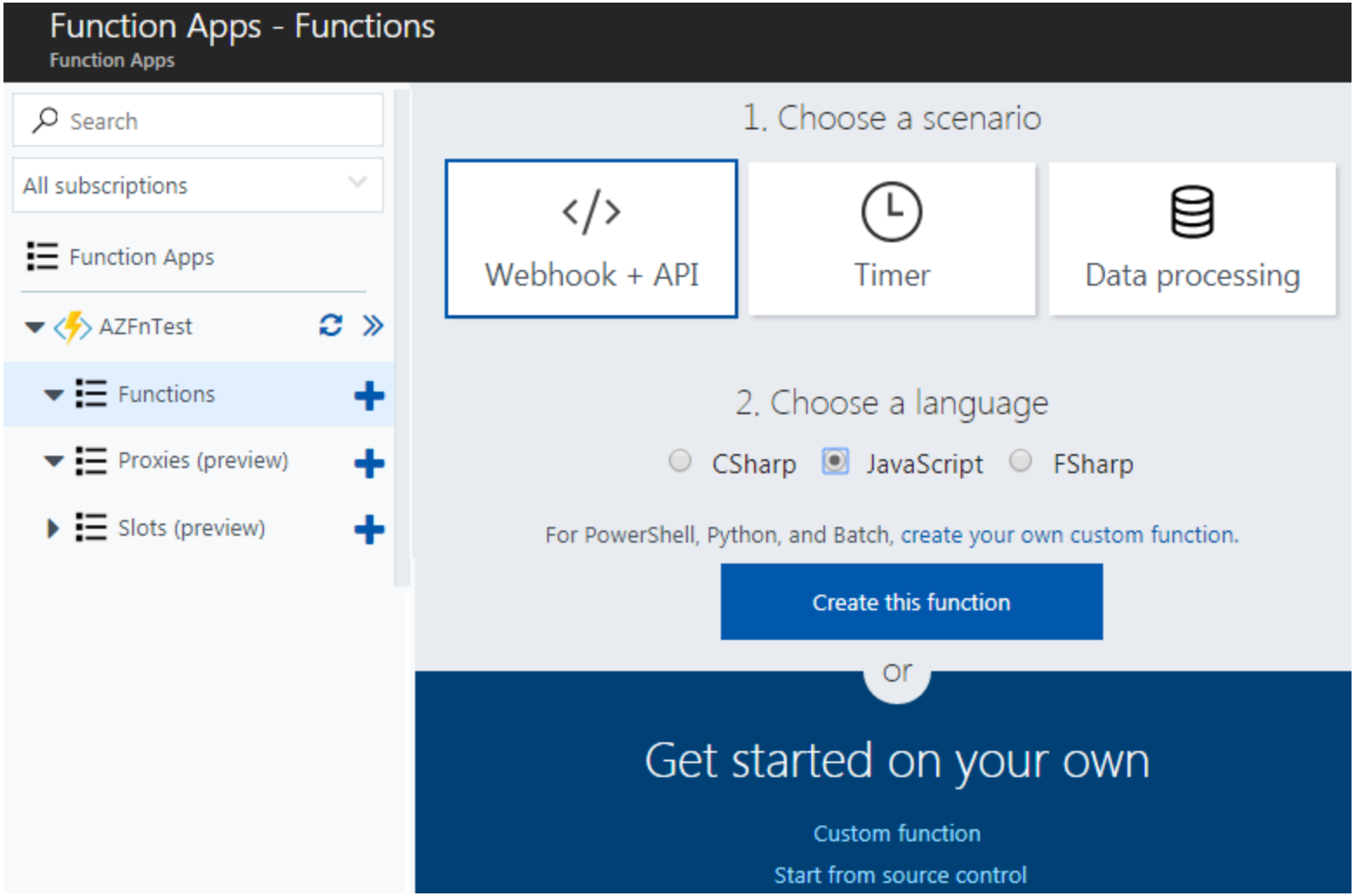
Create Function in Function App

- Click on the **Functions** section in the Function App.
- Select Webhook + API and JavaScript as the language



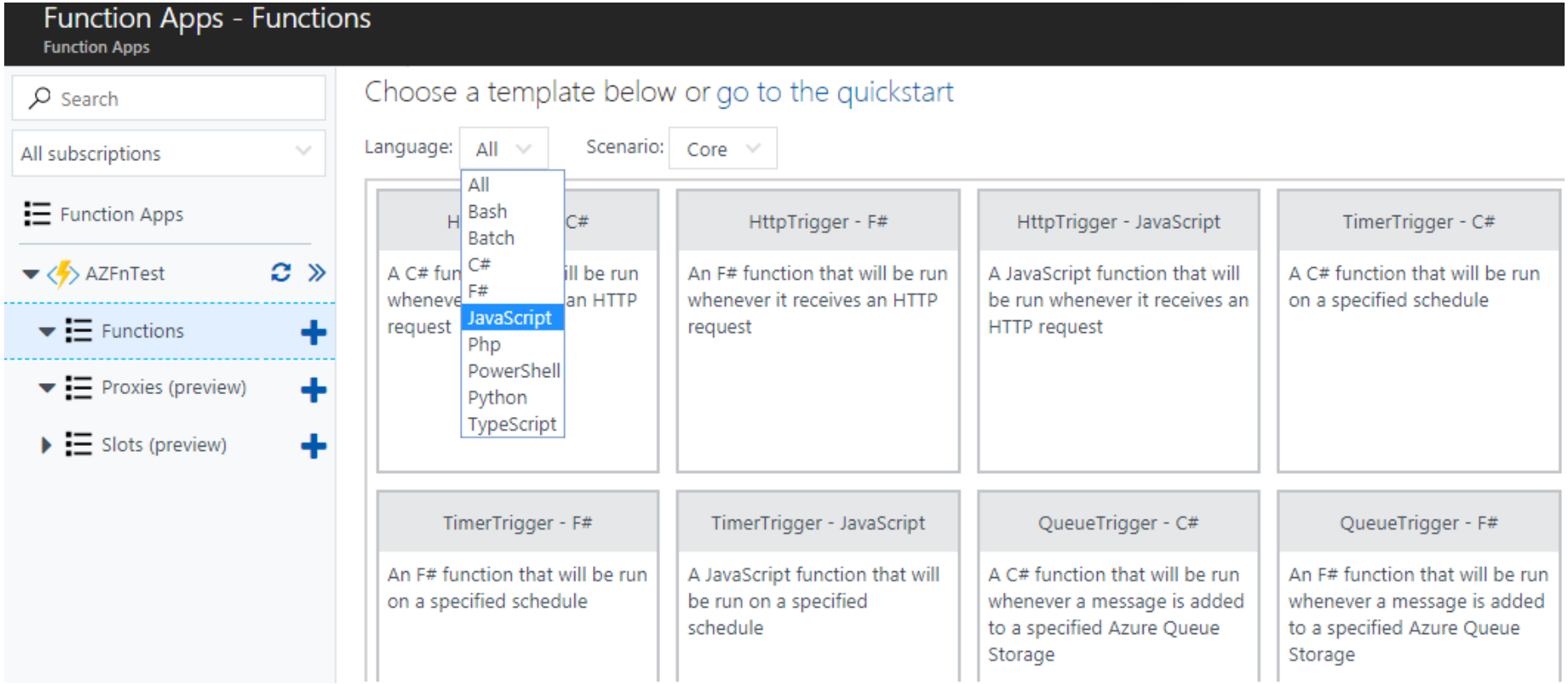
Create Custom Function

- Click on **Custom function** so that we can utilize the already available templates:



Select a Template

- Select JavaScript



Note: Webhooks are "user-defined HTTP callbacks". They are usually triggered by some event, such as pushing code to a repository or a comment being posted to a blog. When that event occurs, the source site makes an HTTP request to the URL configured for the webhook. Users can configure them to cause events on one site to invoke behaviour on another

BlobTrigger Template

- Next, among JavaScript template, select BlobTrigger template:

Function Apps - Functions
Function Apps

Search
All subscriptions
Function Apps
AZFnTest
Functions
Proxies (preview)
Slots (preview)

Choose a template below or go to the quickstart

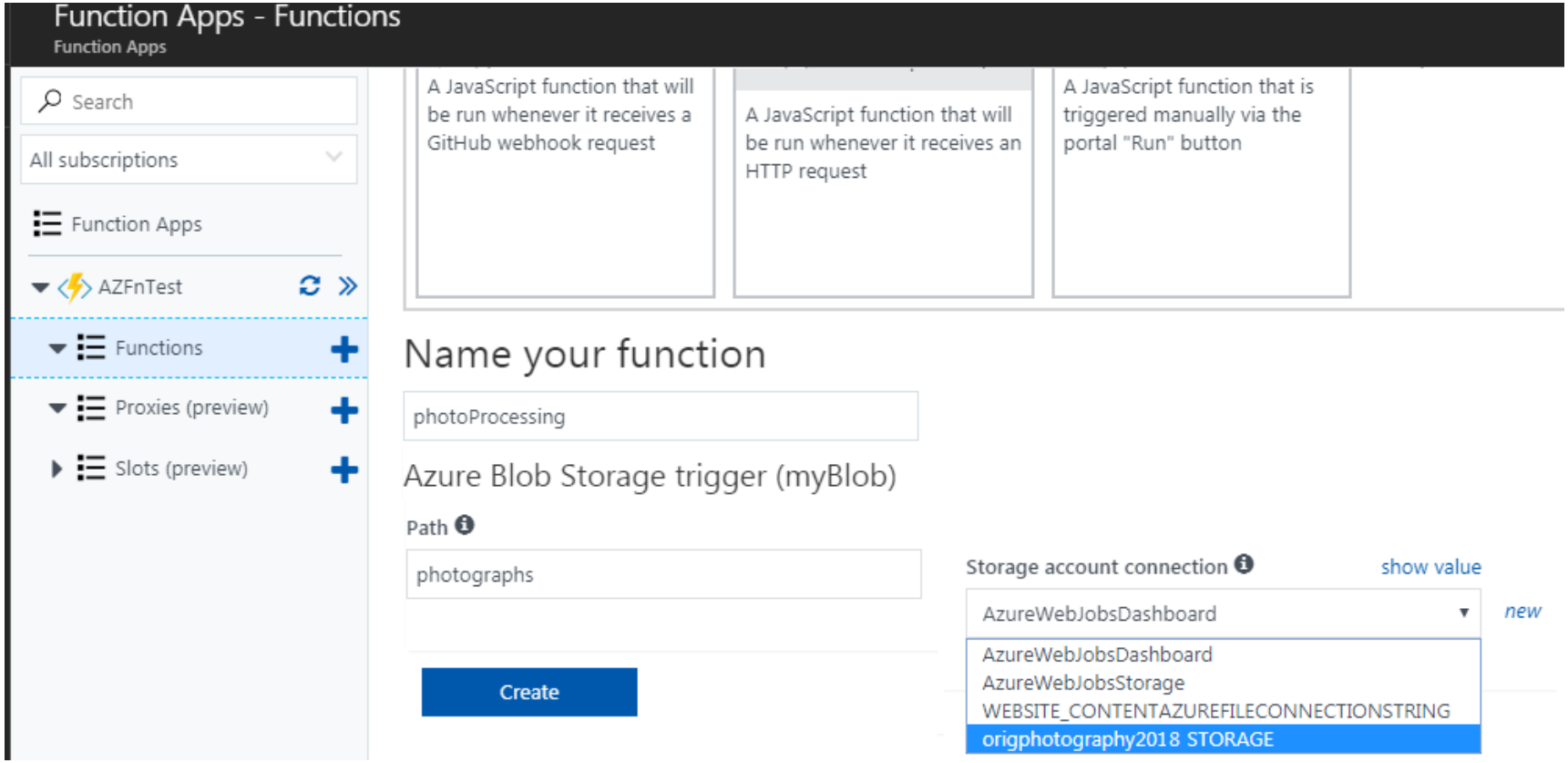
Language: JavaScript Scenario: Core

HttpTrigger - JavaScript	TimerTrigger - JavaScript	QueueTrigger - JavaScript	BlobTrigger - JavaScript
A JavaScript function that will be run whenever it receives an HTTP request	A JavaScript function that will be run on a specified schedule	A JavaScript function that will be run whenever a message is added to a specified Azure Queue Storage	A JavaScript function that will be run whenever a blob is added to a specified container

EventHubTrigger - JavaScript	ServiceBusQueueTrigger - JavaScript	ServiceBusTopicTrigger - JavaScript	Generic Webhook - JavaScript
A JavaScript function that will be run whenever an event hub receives a new event	A JavaScript function that will be run whenever a message is added to a specified Service Bus queue	A JavaScript function that will be run whenever a message is added to the specified Service Bus topic	A JavaScript function that will be run whenever it receives a webhook request

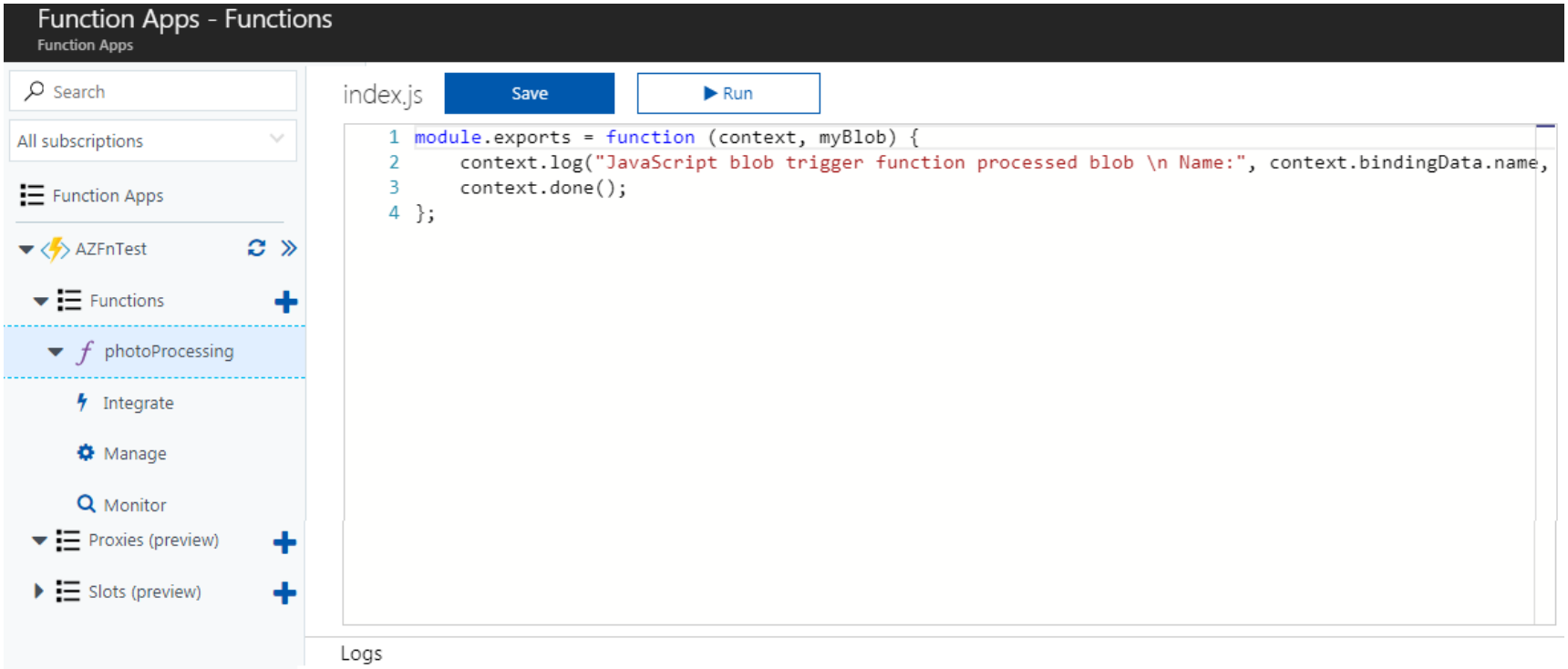
Name your Function

- Provide the name of our function.
- Give the path to the container for the source and select **Storage account connection**:



Code Editor

- Look at the function and code available in the code editor in the Microsoft Azure Portal:



Triggers and Outputs

- Before we write the actual code in the function, let's configure the triggers and outputs. Select **Blob parameter name**, **Storage account connection**, and **Path**.

Function Apps - Functions - Integrate
Function Apps

Search
All subscriptions
Function Apps
AZFnTest
Functions
photoProcess
Integrate
Manage
Monitor
Proxies (preview
Slots (preview)

Advanced editor

Triggers | Inputs | Outputs

Azure Blob Storage (myBlob) | New Input | New Output

Azure Blob Storage trigger (myBlob) delete

Blob parameter name: myBlob

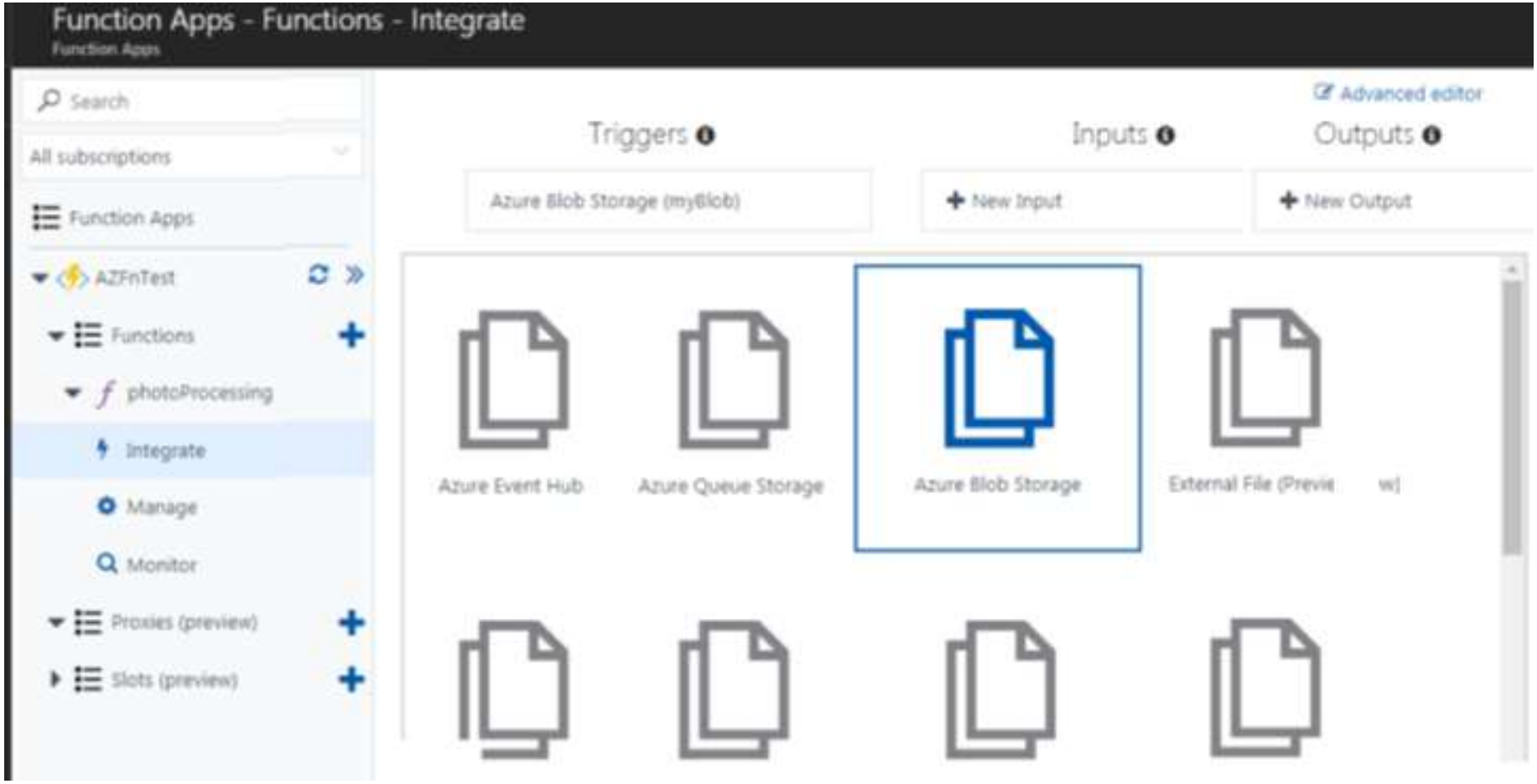
Path: photographs

Storage account connection: origphotography2017_STORAGE show value new

Documentation

Outputs

- Click on **New Output**. Select **Azure Blob Storage**:



Blob Parameters

- Select **Blob parameter name**, **Storage account connection**, and **Path**. Click on **Save**:

Function Apps - Functions - Integrate
Function Apps

"AZFnTest"
All subscriptions
Function Apps
AZFnTest
Functions
Integrate
Manage
Monitor
HttpTriggerForL
Proxies (preview)
Slots (preview)

Triggers
Azure Blob Storage (myBlob)

Inputs
New Input

Azure Blob Storage output (outputBlob)

Blob parameter name
outputBlob

Path
thumbnails/{name}

Use function return value

Storage account connection
show value
origphotography2018_STORAGE
new

Save
Cancel

Review Output Bindings

Function Apps - Functions - Integrate
Function Apps

"AZFnTest"

All subscriptions

Function Apps

AZFnTest

Functions

Integrate

Manage

Monitor

HttpTriggerForLoginAPI

Proxies (preview)

Slots (preview)

Advanced edito

Triggers | Inputs | Outputs

Azure Blob Storage (myBlob) | New Input | Azure Blob Storage ($return) | New Output

Azure Blob Storage output ($return) delete

Blob parameter name

$return

Use function return value

Path

thumbnails/{name}

Storage account connection show value

origphotography2018_STORAGE new

Review Function in Advance Editor

- Click on the advanced editor link to review **function.json**:

Function Apps - Functions - Integrate
Function Apps

"AZFnTest"

All subscriptions

Function Apps

AZFnTest

Functions

Integrate

Manage

Monitor

HttpTriggerForLoginAPI

Proxies (preview)

Slots (preview)

Your app is currently in read\write mode because you've set the edit mode to read\write despite having source control enabled. Any changes you make may get overwritten with your next deployment. To change edit mode visit function app settings.

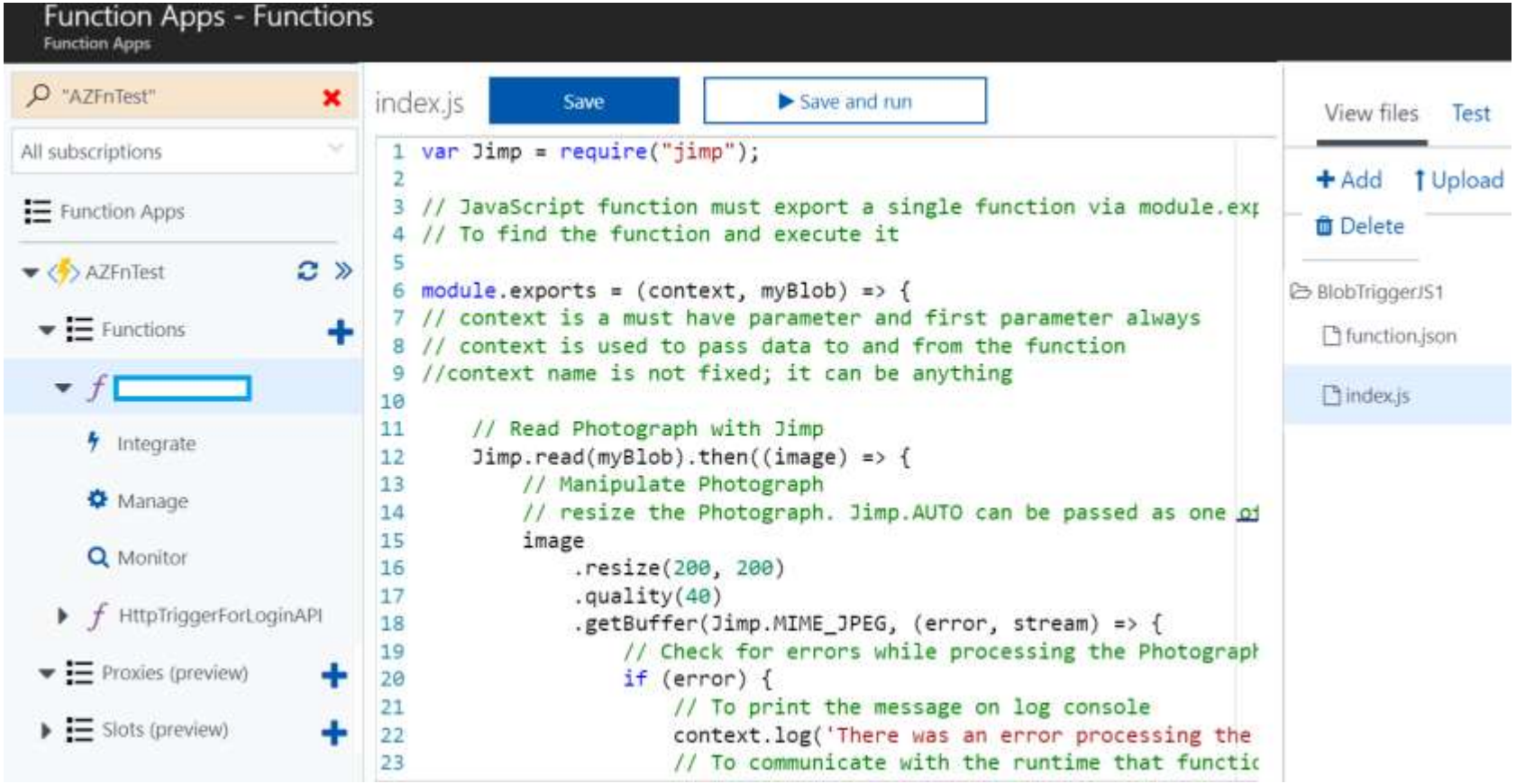
function.json

Standard edi

```
1  {
2    "bindings": [
3      {
4        "name": "myBlob",
5        "type": "blobTrigger",
6        "direction": "in",
7        "path": "photographs/{name}",
8        "connection": "origphotography2017_STORAGE",
9        "dataType": "binary"
10     },
11     {
12       "type": "blob",
13       "name": "$return",
14       "path": "thumbnails/{name}",
15       "connection": "origphotography2017_STORAGE",
16       "direction": "out"
17     }
18   ],
```

Paste your Code into the Editor

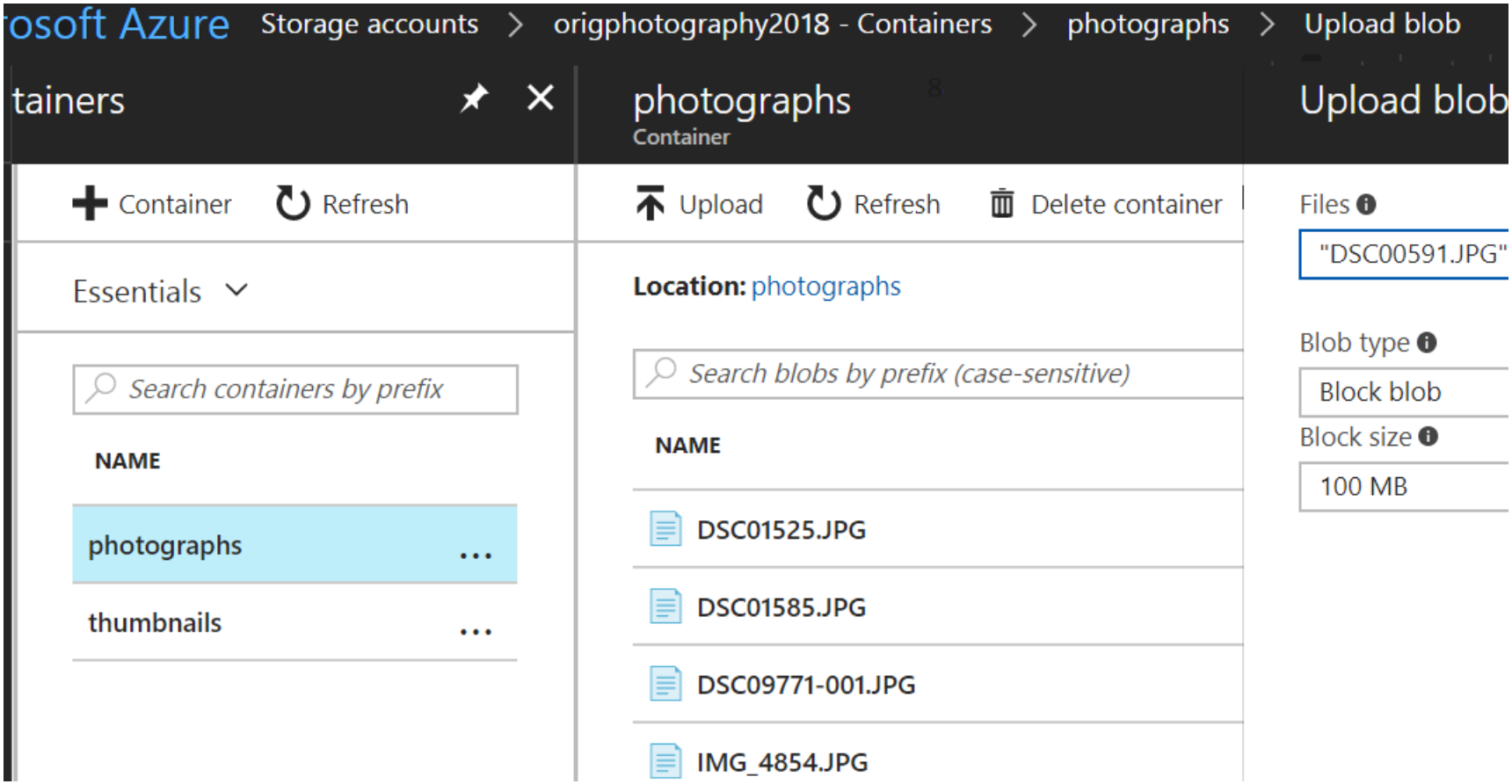
- Now, paste the function code for creating a thumbnail into the **Functions** code editor:



- Now, everything is set and configured, let's upload a photograph in the photographs blob container:

Upload Photograph

- Click on the container and click on **Upload**. Select a photograph to be uploaded to the container.



Check Logs in Function Apps

- Go to **Function Apps** and check the logs.
- We may get an Error: **Cannot find module 'jimp'**:

```
2017-06-30T16:55:01.309 Function started
(Id=411e4d84-5ef0-4ca9-b963-ed94c0ba8e84)
2017-06-30T16:55:01.371 Function completed (Failure,
Id=411e4d84-5ef0-4ca9-b963-ed94c0ba8e84, Duration=59ms)
2017-06-30T16:55:01.418 Exception while executing function:
Functions.photoProcessing. mscorlib: Error: Cannot find module
'jimp'
at Function.Module._resolveFilename (module.js:455:15)
at Function.Module._load (module.js:403:25)
at Module.require (module.js:483:17)
at require (internal/module.js:20:19)
at Object.<anonymous>
```

Troubleshoot Azure Function

- Go to the **Kudu** console of the Azure Function App.
- Click on **Debug Console** and select **Powershell**.
- Execute the npm install -save jimp command:

Kudu | Environment | Debug console | Process explorer | Tools | Site extensions

... / home | 4 items

	Name	Modified
	data	1/22/2018, 5:23:04 PM
	LogFiles	1/22/2018, 5:22:53 PM
	node_modules	1/22/2018, 10:37:15 PM
	site	1/22/2018, 5:22:53 PM

```
Kudu Remote Execution Console
Type 'exit' then hit 'enter' to get a new powershell process.
Type 'cls' to clear the console

PS D:\home> npm install --save jimp
```

jimp Module

- Once the command execution has completed successfully, go to the node_modules directory and review the jimp module in this directory:

Kudu | Environment | Debug console | Process explorer | Tools | Site extensions

... / jimp | 15 items

	LICENSE	2 KB
	omggif.js	30 KB
	package.json	4 KB
	phash.js	6 KB
	README.md	18 KB
	resize.js	17 KB
	resize2.js	12 KB
	server.js	1 KB

```
Kudu Remote Execution Console
Type 'exit' then hit 'enter' to get a new powershell process.
Type 'cls' to clear the console
PS D:\home>
```

Folder Structure

- Understand the folder structure of a function in the Kudu editor. Click on the **Debug Console** and select **Powershell**:

Kudu | Environment | Debug console | Process explorer | Tools | Site exten

Environment

Build	63.60622.2902.0 (ed5d6d7174)
Azure App Service	63.0.8598.17 (rd_websites_stable.170623-1213)
Site up time	00.00:00:03
Site folder	D:\home
Temp folder	D:\local\Temp\

REST API (works best when using a JSON viewer extension)

- App Settings
- Deployments
- Source control info
- Files
- Processes and mini-dumps
- Runtime versions
- Site Extensions: installed | feed
- Web hooks
- WebJobs: all | triggered | continuous
- Functions: list | host config

More information about Kudu can be found on the wiki.

node_modules Directory

- Click on the node_modules directory to review all the modules available for usage:

Kudu | Environment | Debug console | Process explorer | Tools | Site extensions

/ | 4 items

	Name
	data
	LogFiles
	node_modules
	site

```
Kudu Remote Execution Console
Type 'exit' then hit 'enter' to get a new powershell process.
Type 'cls' to clear the console

PS D:\home>
```

Available Modules

- There are many modules available for the Azure Functions to utilize:

Kudu | Environment | Debug console | Process explorer | Tools

... / node_modules | 88 items

	isstream
	jimp
	jpeg-js
	jsbn
	json-schema
	json-stable-stringify
	json-stringify-safe
	jsonify
	isprim

```
Kudu Remote Execution Console
Type 'exit' then hit 'enter' to get a new powershell process.
Type 'cls' to clear the console

PS D:\home>
PS D:\home\node_modules>
```

www.root

- Click on the site in the Kudu editor. Go to www.root and select the function name to review which files are available in the specific function.
- As we saw in the Azure Portal, we can also see `function.json` and `index.js`:

Kudu | Environment | Debug console | Process explorer | Tools | Site extensions

... / photoProcessing | 2 items

	Name	Size
	function.json	1 KB
	index.js	1 KB

```
Kudu Remote Execution Console
Type 'exit' then hit 'enter' to get a new powershell process.
Type 'cls' to clear the console

PS D:\home>
PS D:\home\node_modules>
PS D:\home>
PS D:\home\site>
PS D:\home\site\wwwroot>
PS D:\home\site\wwwroot\photoProcessing>
```

Edit `function.json`

Kudu | Environment | Debug console | Process explorer

Save | Cancel | function.json

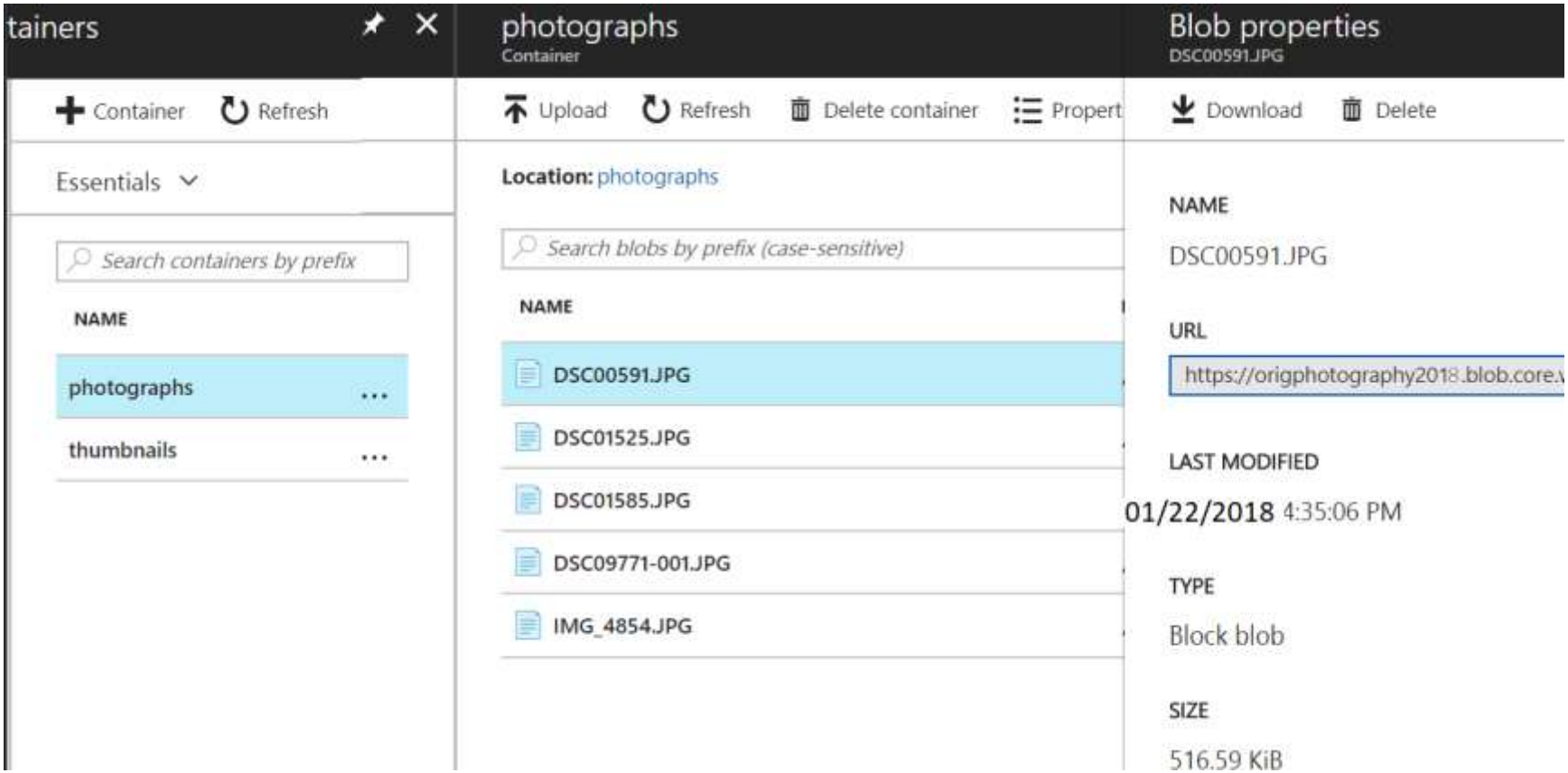
```
1  {
2    "bindings": [
3      {
4        "name": "myBlob",
5        "type": "blobTrigger",
6        "direction": "in",
7        "path": "photographs/{name}",
8        "connection": "origphotography2018_STORAGE",
9        "dataType": "binary"
10     },
11     {
12       "type": "blob",
13       "name": "$return",
14       "path": "thumbnails/{name}",
15       "connection": "origphotography2018_STORAGE",
16       "direction": "out"
17     }
18   ],
19   "disabled": false
20 }
```

Execute Function

- Next, we execute the Azure Function again by uploading a photograph to a blob container.
- We upload a photo to the photographs blob container that we created earlier.
- Once the photograph is uploaded, go to Function Apps and verify the logs:

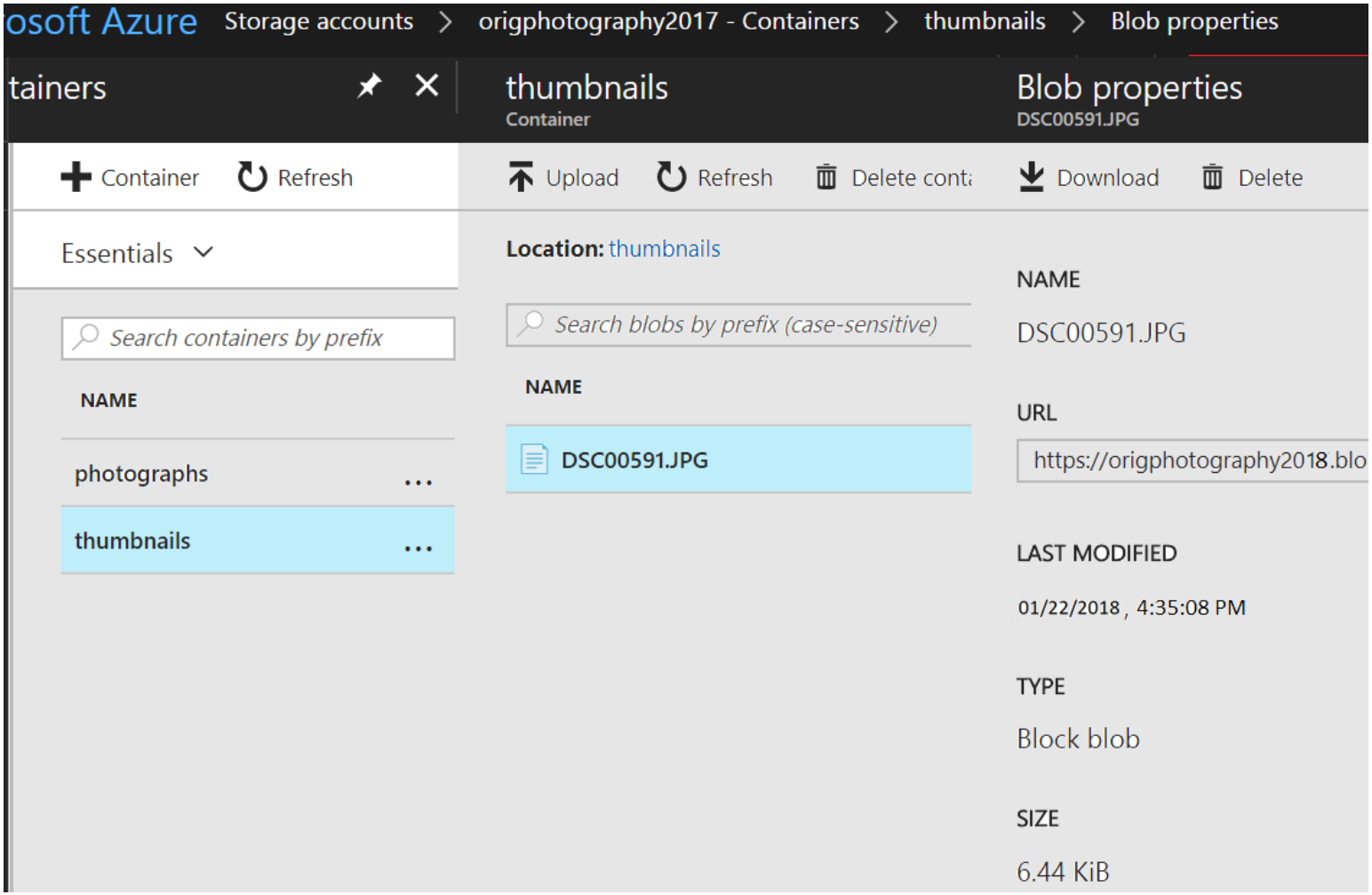
```
2018-01-22T11:03:11 Welcome, you are now connected to log-streaming
service.
2018-01-22T11:04:11 No new trace in the past 1 min(s).
2018-01-22T11:05:11 No new trace in the past 2 min(s).
2018-01-22T11:05:11.656 Function started (Id=e3f715fa-da5b-4cf6-9ada-
410ec8db956a)
2018-01-22T11:06:53.592 Successfully processed the Photograph
2018-01-22T11:06:53.686 Function completed (Success, Id=e3f715fada5b-
4cf6-9ada-410ec8db956a, Duration=102034ms)
```

Check the Photograph is Uploaded



Verify the Thumbnail Container

- Check that the photograph has uploaded in the thumbnails container and verify the size of the photograph:



Triggers

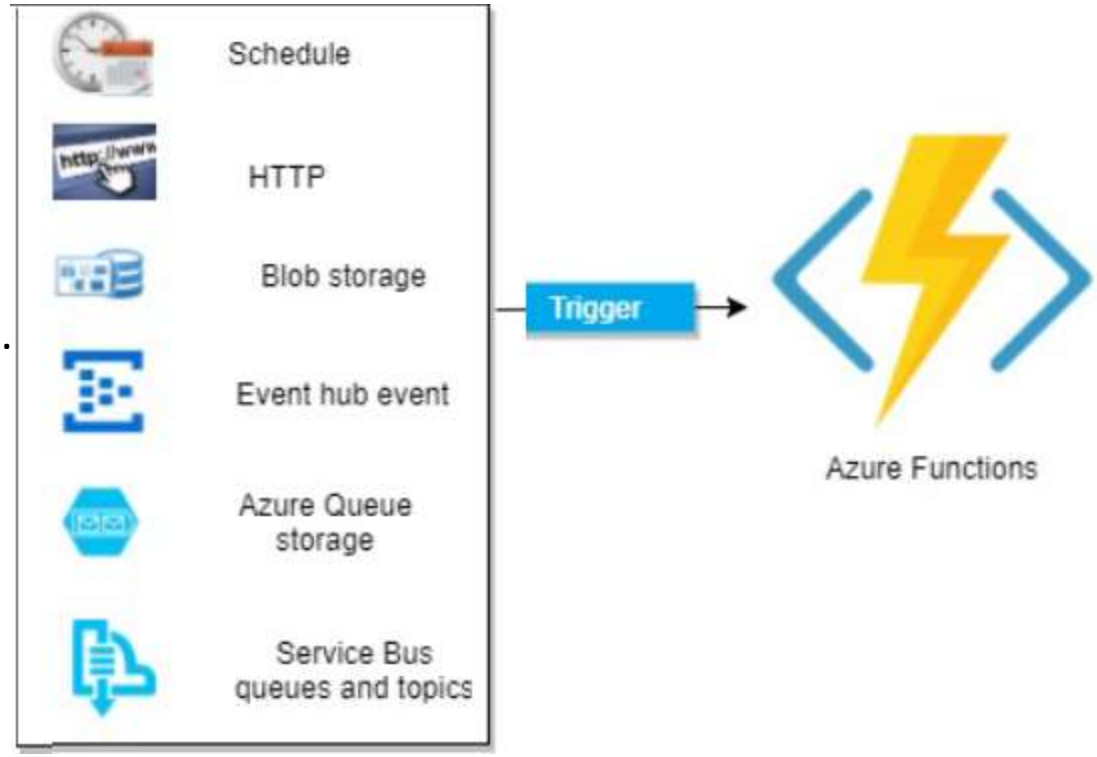
- In normal English, a trigger is an event or situation that causes something to start. This something can be some sort of processing of data or some other service that performs some action.
- Triggers are a set of functions that get executed when some event gets fired.
- Most of us are familiar with Database Triggers. Azure triggers are very similar to database triggers, except that that react to a broader set of events.

- In Azure, we have different types of triggers, such as an implicit trigger, and we can also create a manual trigger.
- Azure Functions allow you to write code in response to a trigger.
- As we have seen on previous slides when a photograph is uploaded into a blob container of a storage account, the Azure Function will start execution and resize the photograph and create a thumbnail of it.
- One Azure function must have exactly one trigger. One function can not have multiple triggers.

Events that fire triggers

Events that fire triggers are:

- scheduled events,
- HTTP messages,
- uploads or deletions in Blob Storage,
- Event Hub Events,
- Azure Queue Messages,
- and Service Buss messages delivered to queues or topics.



Types of Triggers

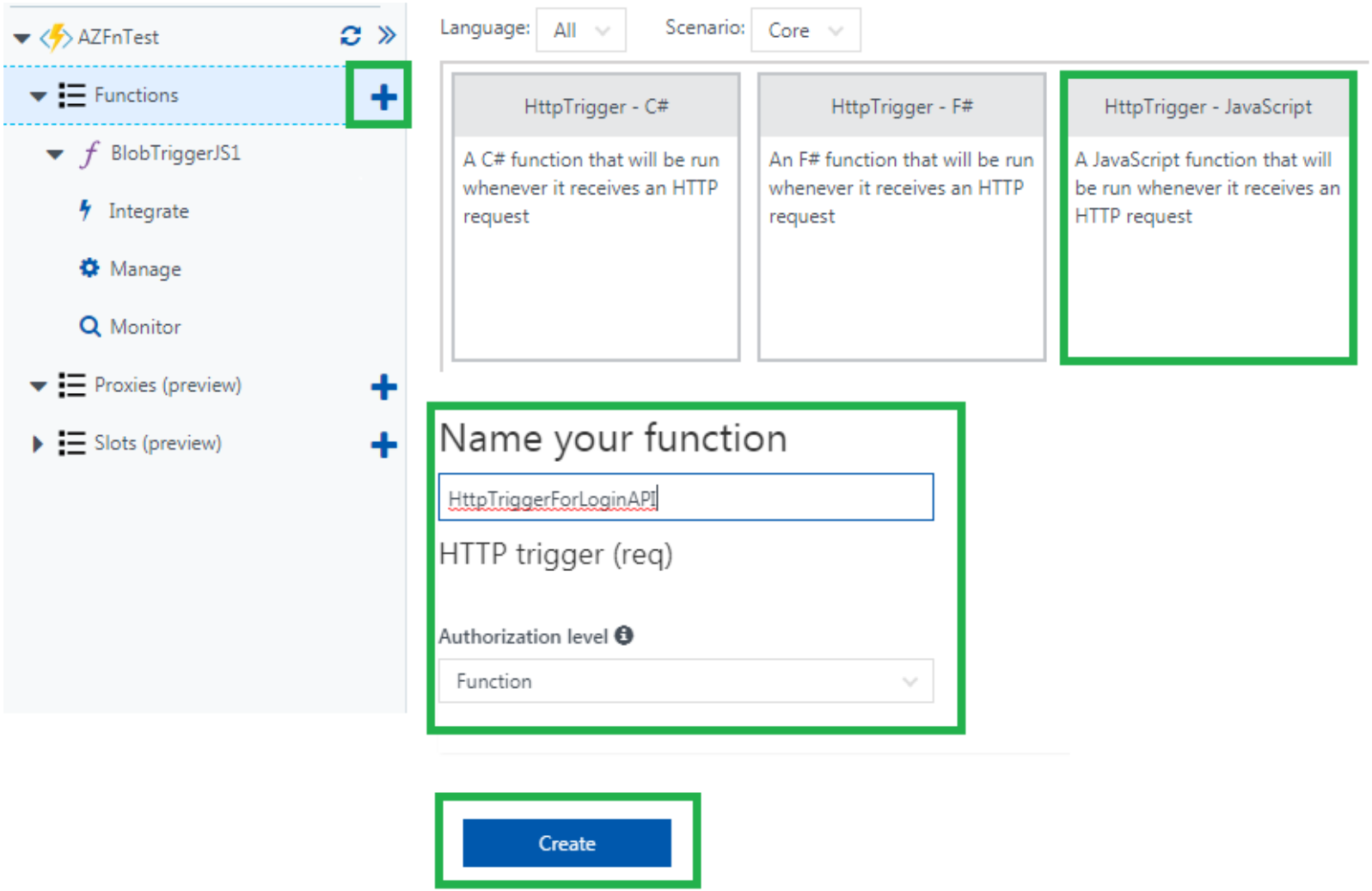
- **TimerTrigger**: This trigger is called on a predefined schedule. We can set the time or execution of the Azure Function using this trigger.
- **BlobTrigger**: This trigger will get fired when a new or updated blob is detected. The blob contents are provided as input to the function.
- **EventHubTrigger**: This trigger is used for the application instrumentation, the user experience, workflow processing, and in the **Internet of Things** (**IoT**). This trigger will get fired when any events are delivered to an Azure event hub.
- **HTTPTrigger**: This trigger gets fired when the HTTP request comes.
- **QueueTrigger**: This trigger gets fired when any new messages come in an Azure Storage queue.
- **Generic Webhook**: This trigger gets fired when the Webhook HTTP requests come from any service that supports Webhooks.
- **GitHub Webhook**: This trigger is fired when an event occurs in your GitHub repositories. The GitHub repository supports events such as Branch created, Delete branch, Issue comment, and Commit comment.
- **Service Bus trigger**: This trigger is fired when a new message comes from a service bus queue or topic.

Http Trigger

- The HTTP trigger is normally used to create the API or services, where we request for data using the HTTP protocol and get the response. We can also integrate the HTTP trigger with a Webhook.
- Now we will create the HTTP Login API. We will send the login credential through an HTTP post request and get the response as to whether the user is valid or not.
- Since we have already created a Function app in the previous example, we can now add multiple functions to it.

Http Trigger Java Script

- Click on **+** |, select **HttpTrigger-JavaScript,** provide the function name, and click on the **Create** button:



Default Template

- After we click on the **Create** button, the default template will be available. Now, we can edit and test the function:

index.js | Save | ▶ Run | </> Get function URL | View files | Test

```
1  module.exports = function (context, req) {
2      context.log('JavaScript HTTP trigger function processed a request.');
3
4      if (req.query.name || (req.body && req.body.name)) {
5          context.res = {
6              // status: 200, /* Defaults to 200 */
7              body: "Hello " + (req.query.name || req.body.name)
8          };
9      }
10     else {
11         context.res = {
12             status: 400,
13             body: "Please pass a name on the query string or in the request boc
14         };
15     }
16     context.done();
17 };
```

HTTP method

POST

Query

There are no query parameters

+ Add parameter

Headers

There are no headers

+ Add header

Request body

```
1 {
2     "name": "Azure"
3 }
```

Logs | Pause | Clear | Copy logs | Expand

Edit the Code

- Now edit the code as follows:

```
module.exports = function (context, req) {
    context.log('JavaScript HTTP trigger function processed a request.');
    if (req.body && req.body.username && req.body.password) {
        if(req.body.username != 'admin' && req.body.password != '@dm!n1'){
            context.res = {
                body: "Invalid user"
            };
        }else{
            context.res = {
              body: "User" + (req.body.username)+" is valid"
            };

        }
    }
    else {
        context.res = {
            status: 400,
            body: "Please provide a username and password in the request body"
        };
    }
    context.done();
};
```

Save and Run

- Save and run the code, as shown in the following screenshot:

View files Test

Request body

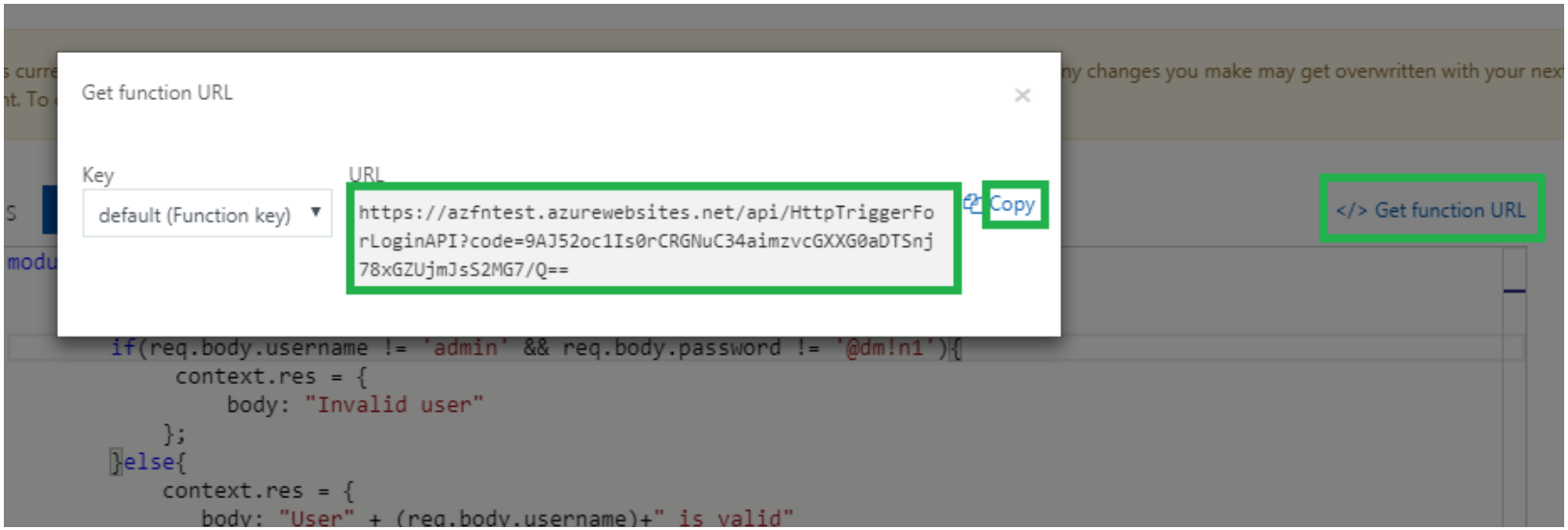
```
1 {
2     "username": "admin",
3     "password": "@dm!n1",
4 }
```

Output Status: 200 OK

```
"Useradmin is valid"
```

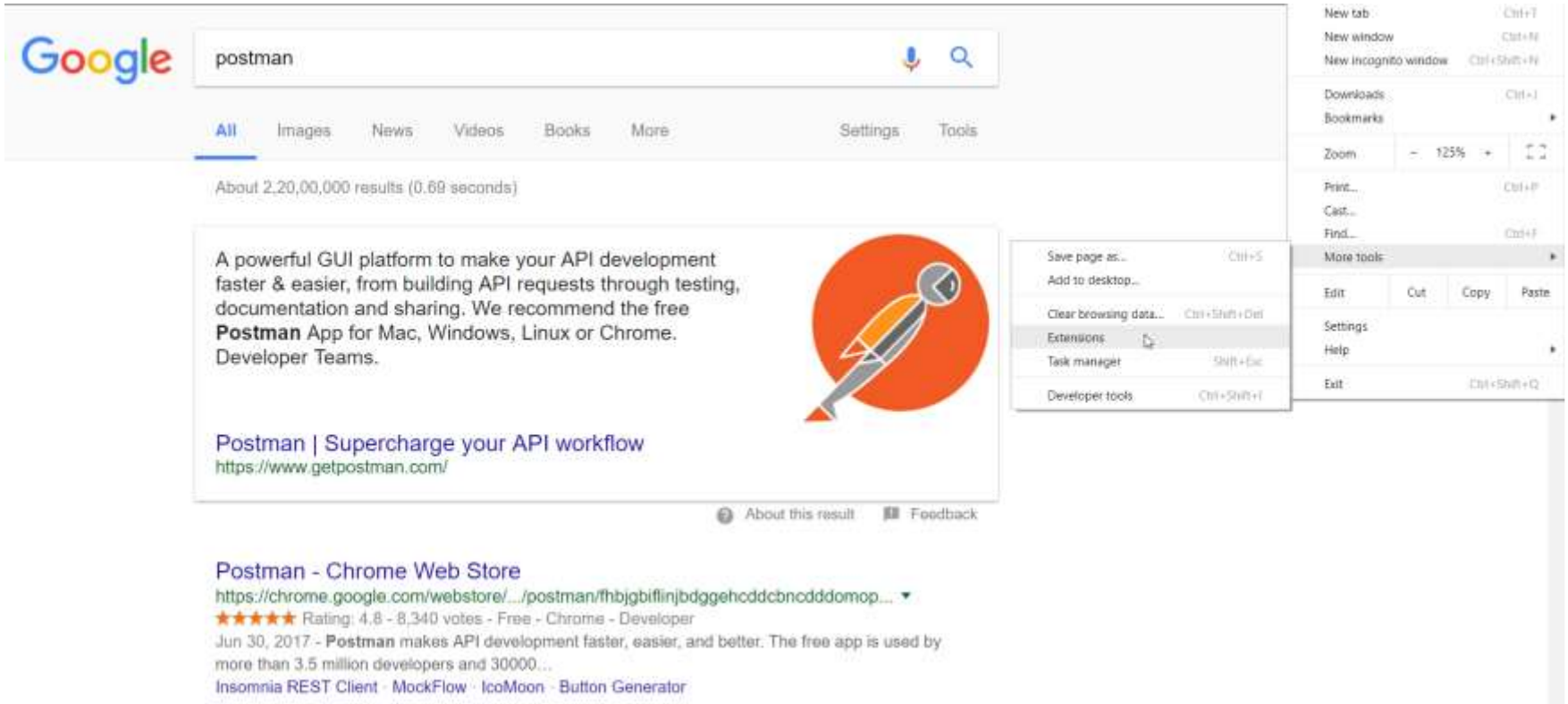
Check the Service in Postman

- Check this service in Postman. To get the **URL** from the function, click on **Get function URL**:



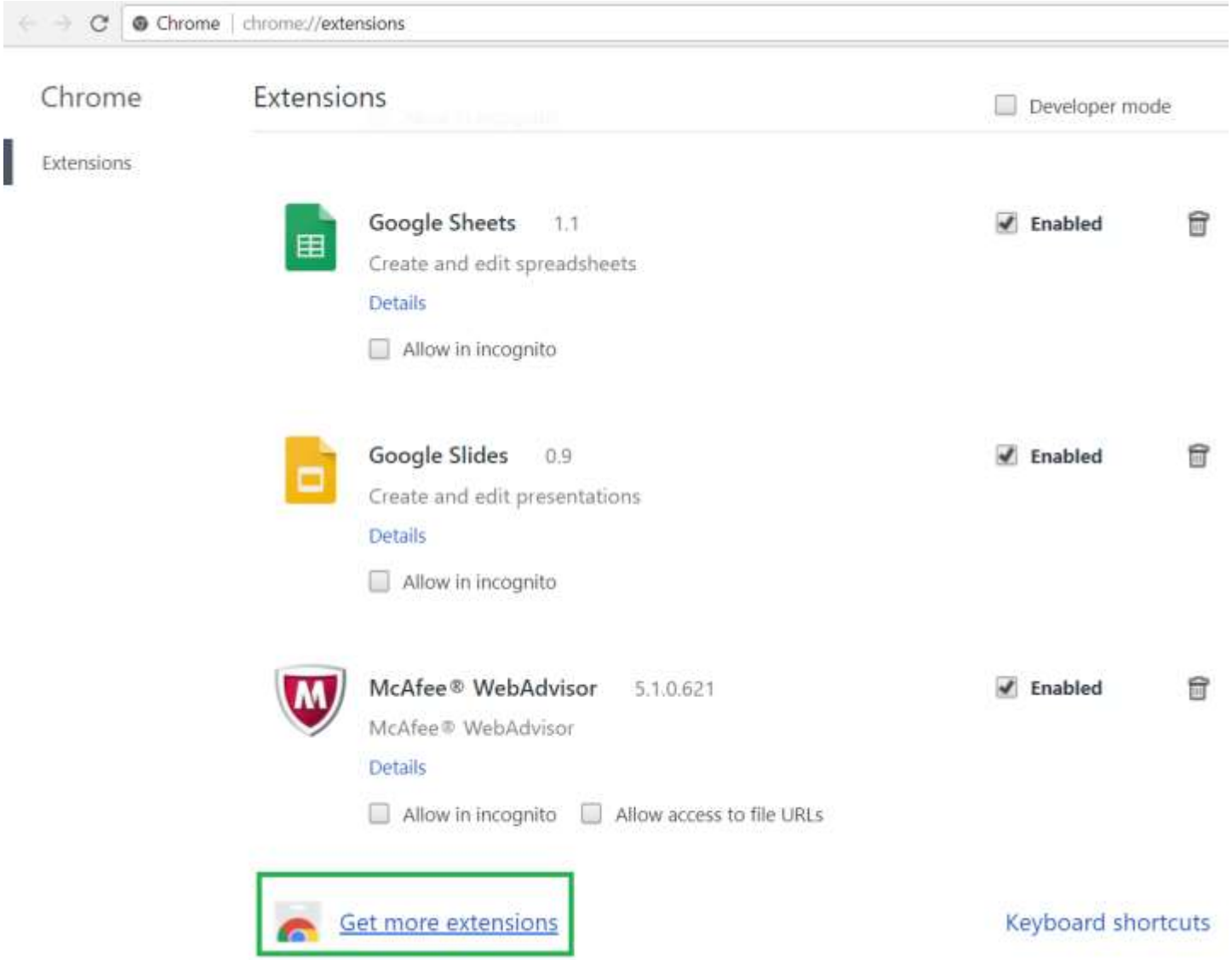
Postman

Postman is a Chrome extension for REST developers to test APIs. To add the Chrome extension, go to **Settings** in Chrome and select **More tools** | **Extensions,** as shown in the following:



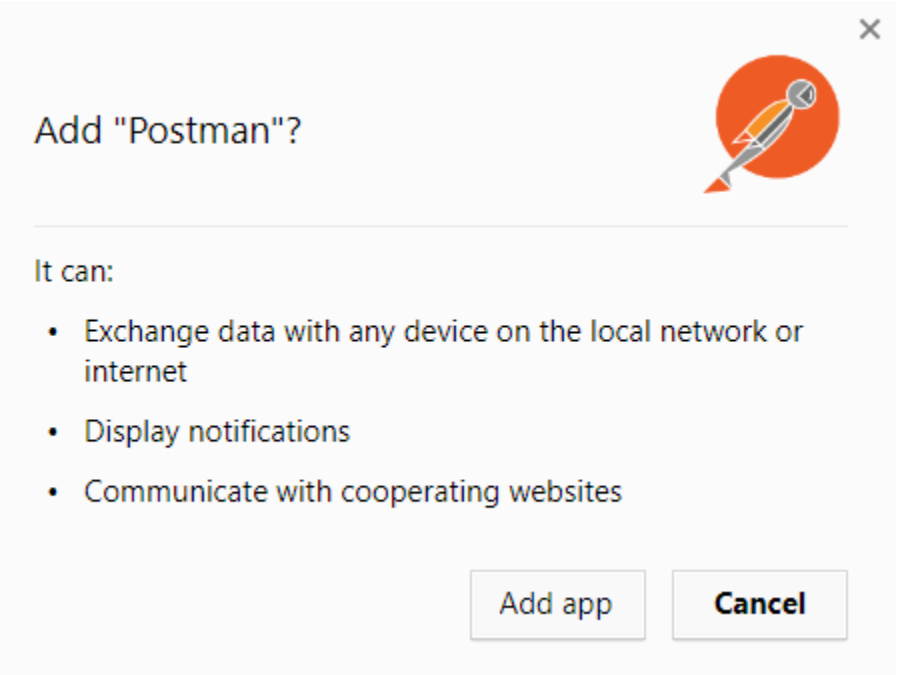
Extensions

- Click on **Get more extensions**:



Add Postman

- Now search for postman and then click on **+ ADD TO CHROME**.
- Click on **Add app**.



- Launch the Postman app. Click on **Sign Up with Google**.
- Copy the function URL and paste it in Postman. Select the method type **POST** and provide a request body and click on the **Send** button:

Trigger Response

- If we provide the correct username and password in the request body, we will get the response, **user is valid; otherwise,** the response will be **invalid user**.

POST https://azfntest.azurewebsites.net/api/HttpTriggerForLoginAPI?code=9AJ52oc1I Params Send

Authorization Headers (1) Body Pre-request Script Tests

form-data x-www-form-urlencoded raw binary JSON (application/json)

```
1 {
2     "username": "admin",
3     "password": "@dm!n1"
4 }
```

Body Cookies Headers (11) Tests Status: 200 OK

Pretty Raw Preview JSON

```
1 "Useradmin is valid"
```

Bindings

Functions can output their results in several ways:

- Through available input and output bindings
- Event Bus/Service Bus
- Storage

- In the Azure Function, binding is used to bind other resources of Azure with our Azure Function.
- For example, you want to load an external configuration file from blob storage when your function starts. You need to bind the file with the Azure Function.
- Using binding, you do not need to hardcode anything in your code. You just need to bind the configuration file with the function and take the file from the blob storage when it starts.
- We can bind not only the input of the function but also the output of the function too.
- For example, after executing the function we need to store the result in the Azure Table Storage. For that, we need our function output bind with Azure Table Storage.
- A function can have multiple input and output bindings and bindings are optional.

Types of Input Bindings

There are four types of input bindings:

- **Blob storage**: Blob content is used as input to the Azure Function. For example, in the previous scenario where we wanted to create thumbnails for an image whenever a new image is uploaded to the blob storage. In this case, we will create a blob trigger with input bind and blob storage.
- **Storage tables**: Storage table content is used as input to the Azure Function. For example, instead of hardcoding configuration data in the Azure Function, store all of the configuration in the storage table and bind with the Azure Function. When your function runs, it takes all the input from the storage table.
- **SQL tables**: SQL table data can also be used as input for the Azure Function. For example, we want to check the quantity of a product at the end of every day. We have all the product details stored in the SQL table. We have the Azure Function, which is bound with the SQL table and triggers at the end of every day. Once the Azure Function runs, it takes the input from the SQL table and processes the data.
- **NoSQL DB**: No-SQL data like data, which is stored as a document, can also be used as input to the Azure Function. For example, the Azure Function reads JSON data which is stored in NoSQL DB and processes it.

Types of Output Bindings

There are nine types of output bindings:

- **HTTP (REST or Webhook)**: The Azure Function can produce an output as HTTP (REST or Webhook). For example, the output of the Azure Function can be linked to Webhook or can be HTTP REST.
- **Blob storage**: The Azure Function can use blob storage for output binding. For example, we have the Azure Function, which processes the image and compresses the size of image. After the image is compressed, we want it to be stored in blob storage. For this, we need output binding for the Azure Function with blob storage.
- **Events**: An event can also be bound as output from the Azure Function.
- **Queues and topics**: Queues and topics can also be an output from the Azure Function.
- **Storage tables**: Storage tables can also be bound as output to the Azure Function. Storage tables can be used as input or output to the Azure Function. For example, the Azure Function can take input from the storage table, process the data, and after processing again, store data in the storage table.
- **SQL tables**: SQL table can be used as input or output to the Azure Function. The Azure Function can read or take input from the SQL table and after processing again, can store data to the SQL table.
- **NoSQL DB**: No-SQL tables can also be used as input or output to the Azure Function. The Azure Function can read data from NoSQL DB and process it. Once the data is processed, it can be stored in NoSQL DB.
- **Push notifications**: Push notifications can be used as output binding for the Azure Function. For example, we want to send a push notification with a confirmation message to all users who have registered for some event.