KodeKloud

Visit www.kodekloud.com to learn more.

Exploring Azure Functions

Introduction

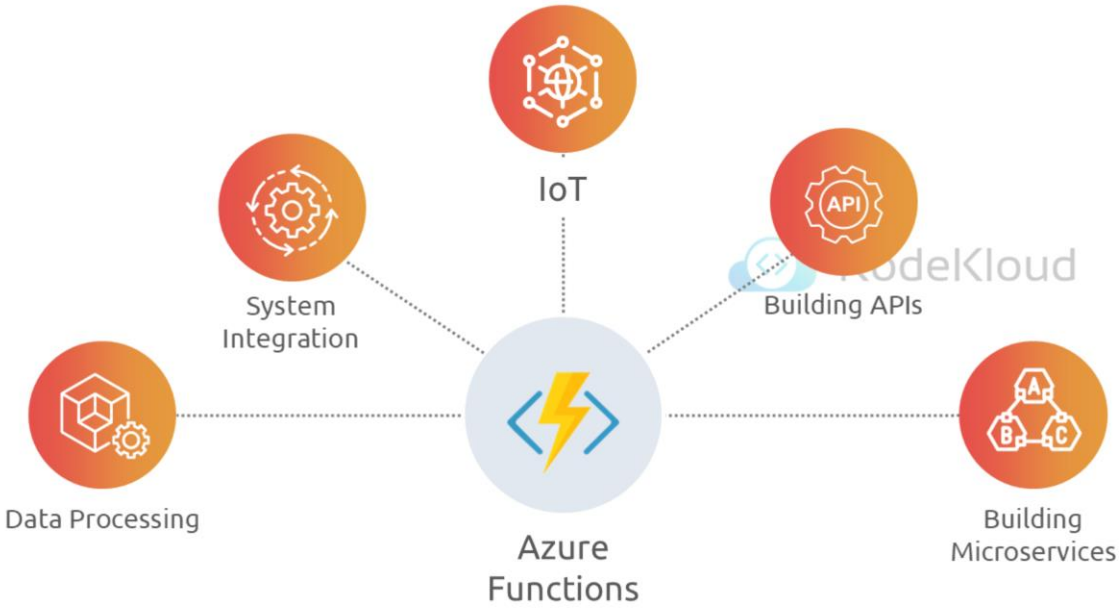
01 Azure Functions is a serverless solution

02 Azure Functions can scale with demand

03 The cloud infrastructure provides all the up-to-date resources needed to keep your applications running

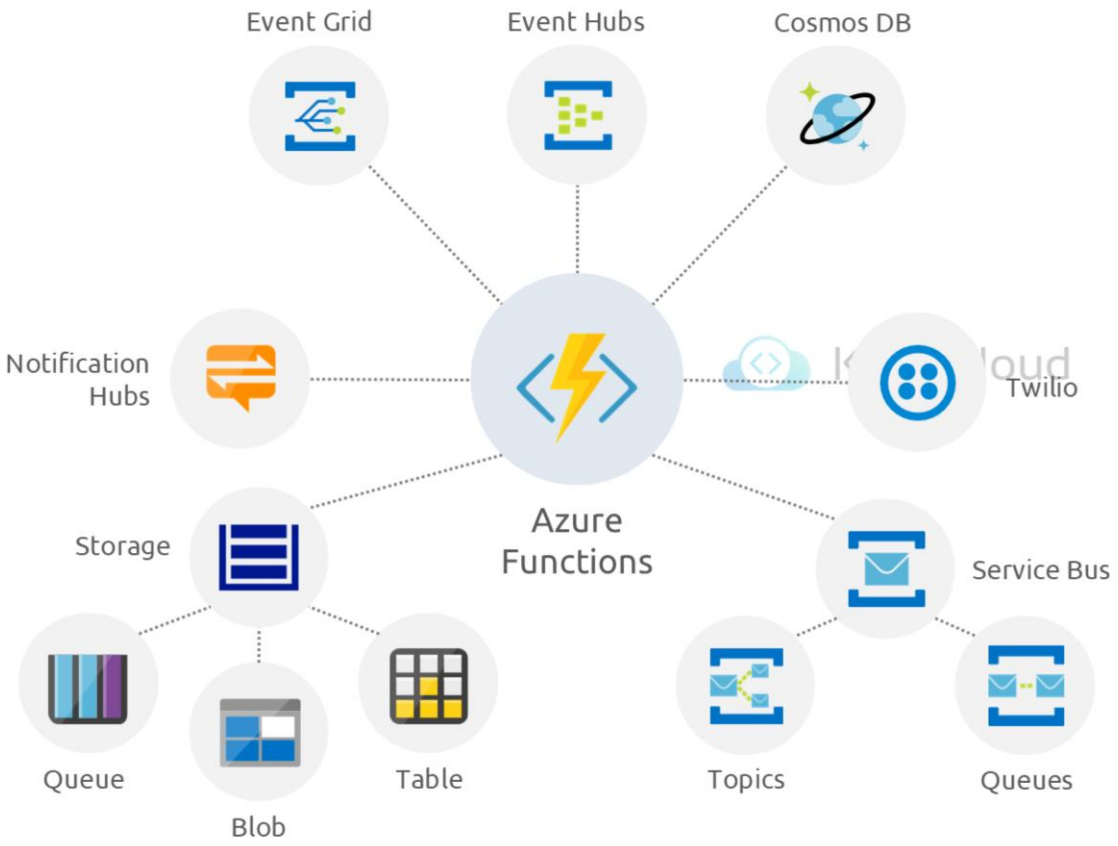
Azure Functions

Azure Functions



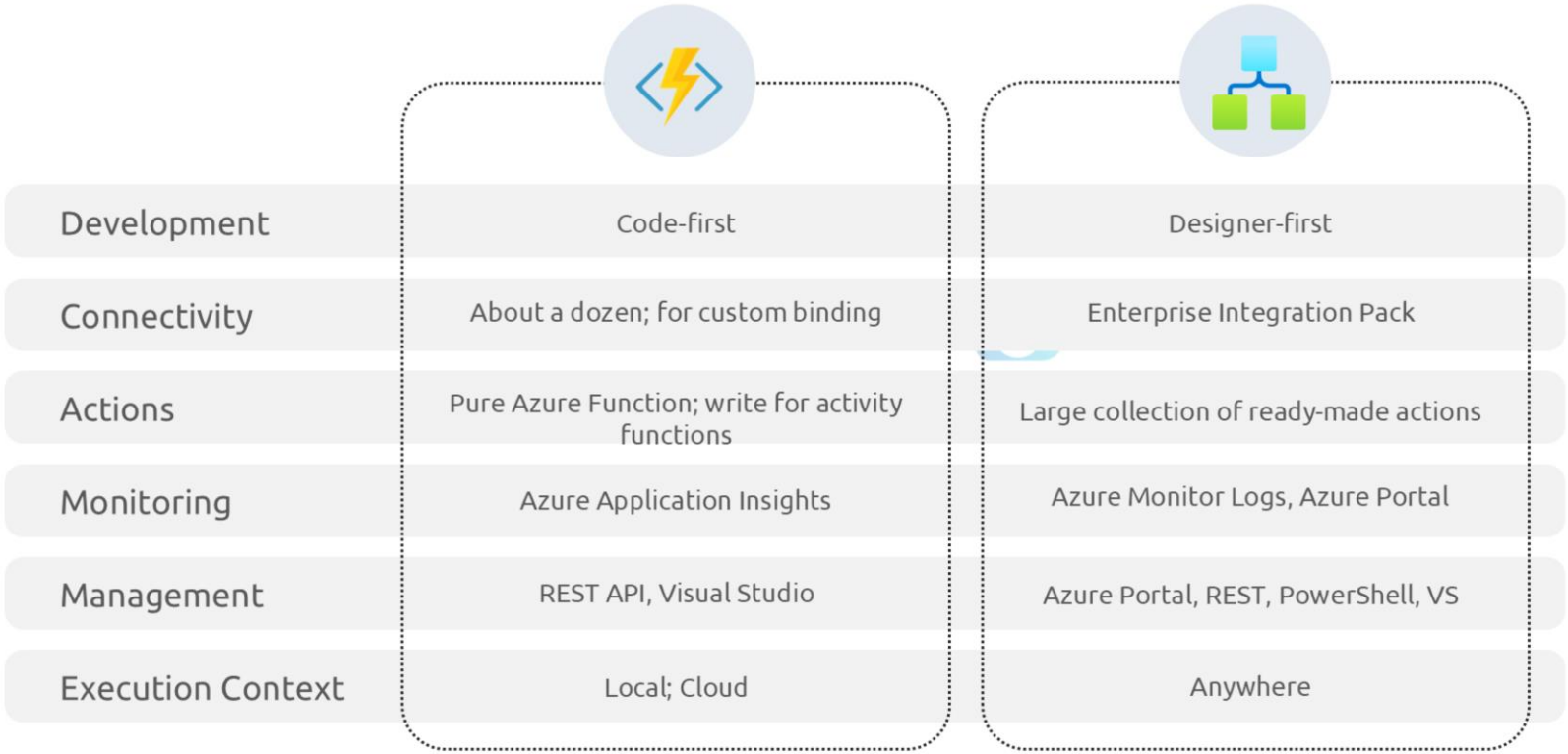
Consider Functions for tasks like image or order processing, file maintenance, or for any tasks that you want to run on a schedule.

Azure Functions



Azure Functions supports triggers, which start execution of your code, and bindings, which simplify coding for input and output data.

Azure Functions vs Logic Apps

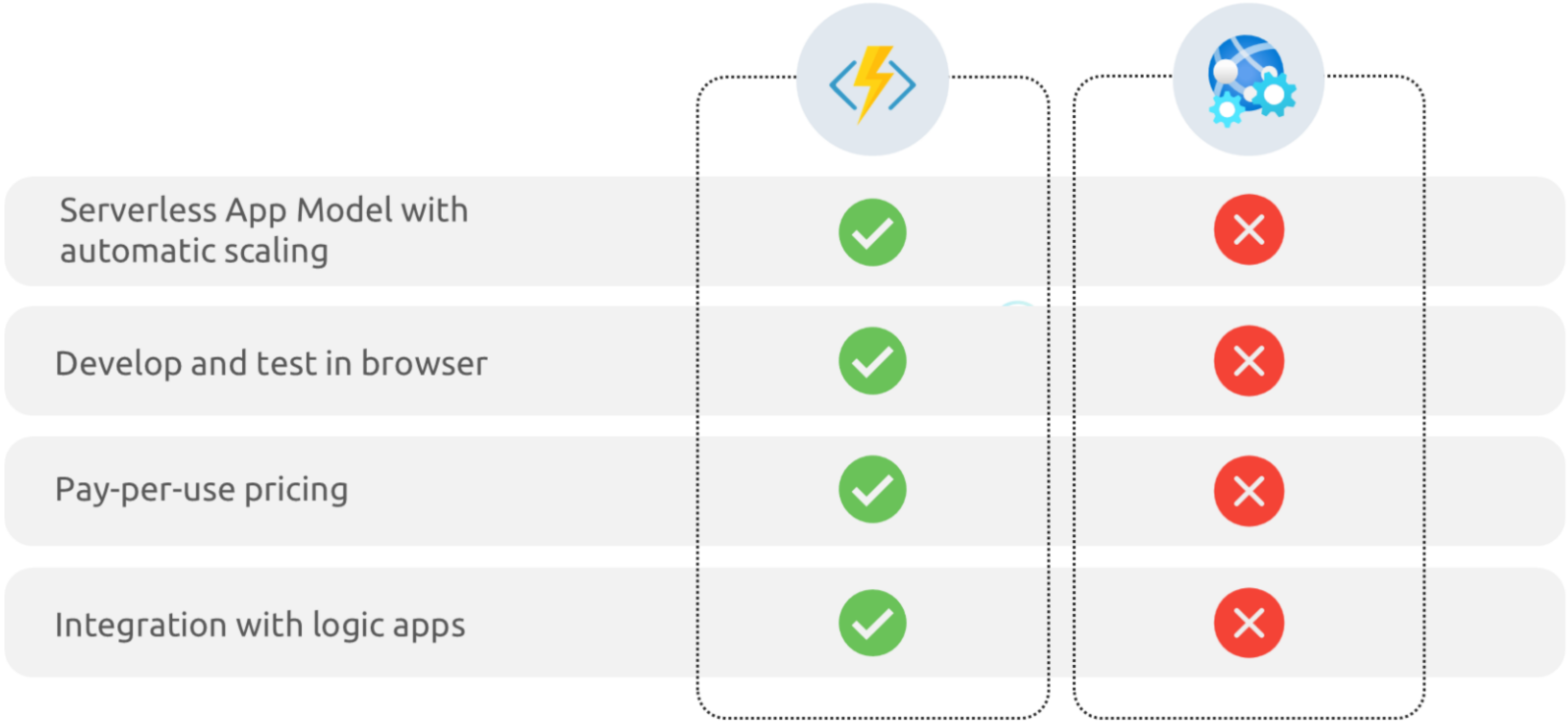


Both Functions and Logic Apps enable serverless workloads. Azure Functions is a serverless compute service, whereas Azure Logic Apps provides serverless workflows. Both can create complex orchestrations. An orchestration is a collection of functions or steps, called actions in Logic Apps, that are executed to accomplish a complex task.

For Azure Functions, you develop orchestrations by writing code and using the Durable Functions extension. For Logic Apps, you create orchestrations by using a GUI or editing configuration files.

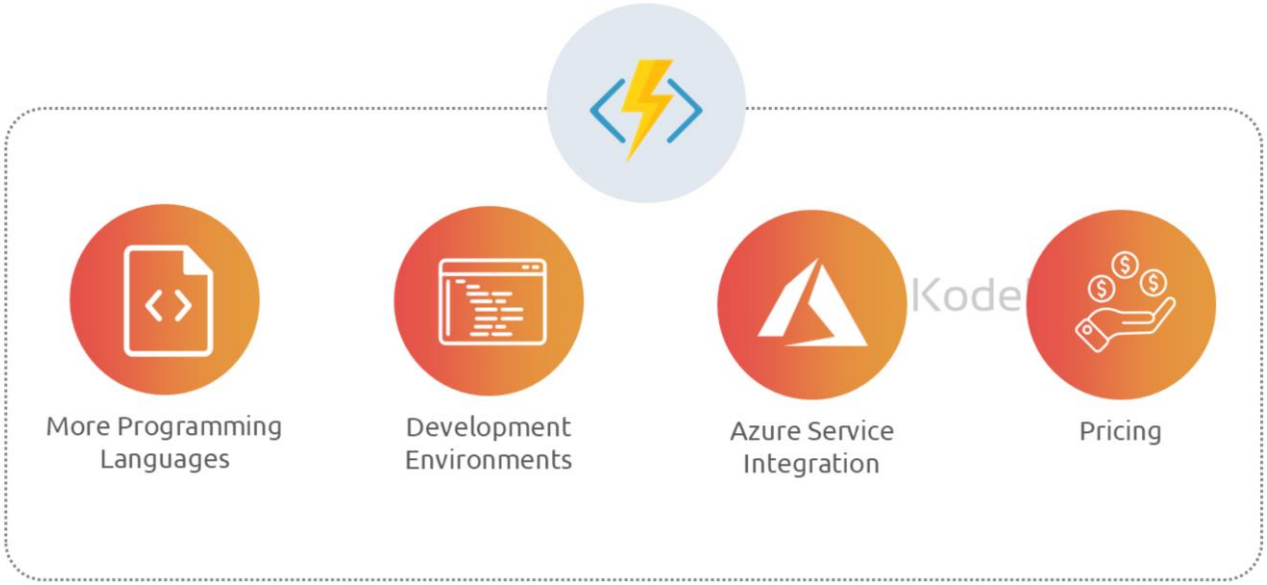
You can mix and match services when you build an orchestration, calling functions from logic apps and calling logic apps from functions.

Azure Functions vs WebJobs



Azure Functions offers more developer productivity than Azure App Service WebJobs does. It also offers more options for programming languages, development environments, Azure service integration, and pricing. For most scenarios, it's the best choice.

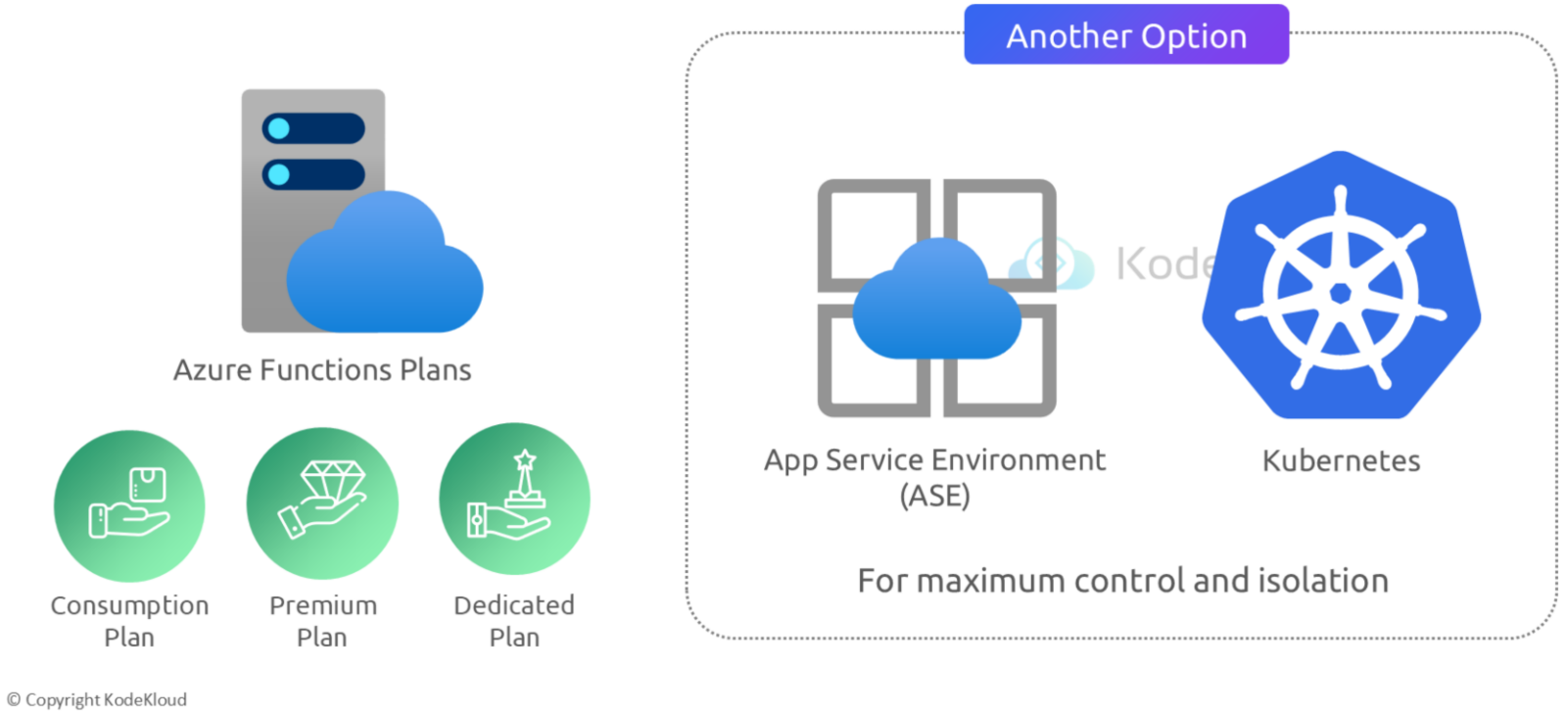
Azure Functions vs WebJobs



Azure Functions offers more developer productivity than Azure App Service WebJobs does. It also offers more options for programming languages, development environments, Azure service integration, and pricing. For most scenarios, it's the best choice.

Azure Functions Hosting Options

Azure Functions Hosting Options



There are three basic hosting plans available for Azure Functions:

- Consumption plan
- Premium plan
- Dedicated plan (App Service)

Additional options for maximum control and isolation:

- App Service Environment

Kubernetes

ASE - App Service Environment (ASE) is an App Service feature that provides a fully isolated and dedicated environment for securely running App Service apps at high scale.
Kubernetes - Kubernetes provides a fully isolated and dedicated environment running on top of the Kubernetes platform.

Hosting plans dictate:
How your function app is scaled.
The resources available to each function app instance.
Support for advanced functionality, such as Azure Virtual Network connectivity.

Hosting Plans Dictate...

01

How to scale function app

02

How to identify available resources

03

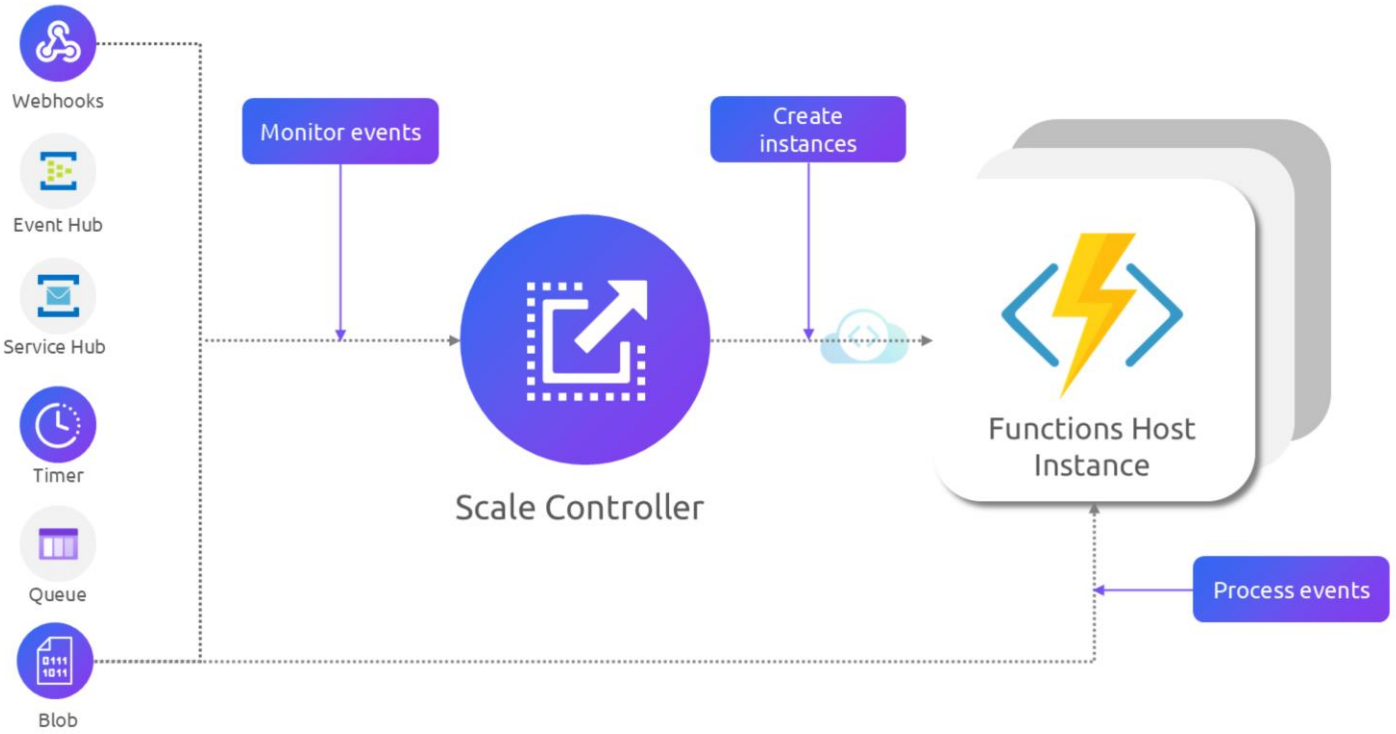
How to support advanced functionalities

Azure Functions Scaling

Scaling Azure Functions

1. Function app is the unit of scale for Azure Functions.
2. A scale controller monitors scaling out or scaling in.
3. Instances may be "scaled in" to zero when no functions are running.
4. The next request has a cold start scaling from zero to one.

Scaling Azure Functions



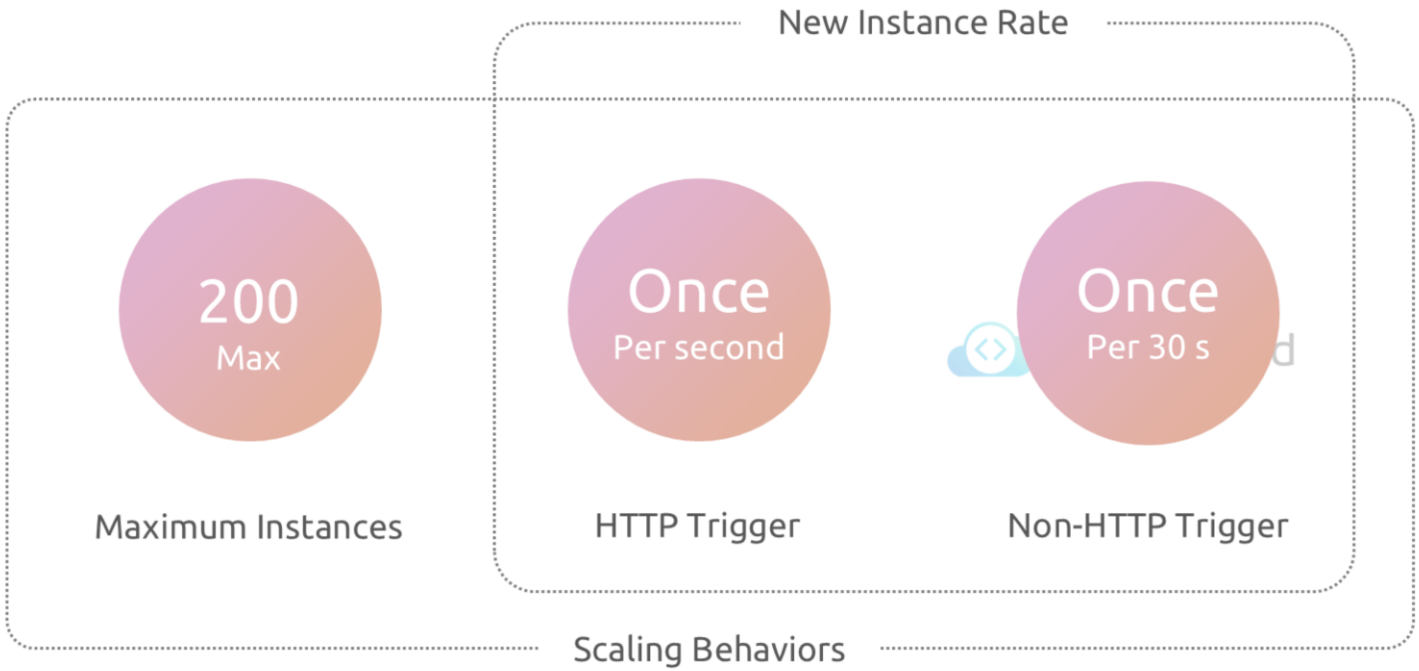
Scaling Azure Functions

Plan	Scale out	Max instances
Consumption	Event driven	Windows:200 Linux: 100
Premium	Event driven	Windows: 100 Linux: 20-100
Dedicated	Plan dependent	10-30, 100 for ASE
Container Apps	Event driven	10-300

Scaling Azure Functions

Plan	Default timeout	Max time
Consumption	5	10
Premium	30	Unlimited
Dedicated	30	Unlimited
Container Apps	30	Unlimited

Scaling Azure Functions



Scaling behaviors
Scaling can vary on a number of factors, and scale differently based on the trigger and language selected. There are a few intricacies of scaling behaviors to be aware of:
Maximum instances: A single function app only scales out to a maximum of 200 instances. A single instance may process more than one message or request at a time though, so there isn't a set limit on number of concurrent executions.
New instance rate: For HTTP triggers, new instances are allocated, at most, once per second. For non-HTTP triggers, new instances are allocated, at most, once every 30 seconds.

Limit scale out

Limit scale out
You may wish to restrict the maximum number of instances an app used to scale out.
This is most common for cases where a downstream component like a database has limited throughput.

By default, Consumption plan functions scale out to as many as 200 instances, and Premium plan functions will scale out to

as many as 100 instances. You can specify a lower maximum for a specific app by modifying the functionAppScaleLimit value. The functionAppScaleLimit can be set to 0 or null for unrestricted, or a valid value between 1 and the app maximum.

Scaling Azure Functions

Limit Scale Out

Authentication Required

Developing Azure Functions

Introduction

01 Functions enable seamless integration with your preferred code editor and development tools for creating and testing code locally on your computer.

02 Local functions can connect to live Azure services, allowing for debugging on your local machine with the complete Functions runtime.

Exploring Azure Functions Development

Azure Functions Development

01

The function.json file defines the configuration settings

02

A function has only one trigger

03

The config file to determine the events to monitor

04

The bindings property is where you configure


```
{
  "disabled":false,
  "bindings":[
      // ... bindings here
      {
          "type": "bindingType",
          "direction": "in",
          "name": "myParamName",
          // ... more depending on binding
      }
  ]
}
```

Each binding shares a few common settings and some settings which are specific to a particular type of binding. Every binding requires the following settings:
type / string - Name of binding. For example, queueTrigger.
direction / string - Indicates whether the binding is for receiving data into the function or sending data from the function. For example, in or out.
name / string - The name that is used for the bound data in the function. For example, myQueue.

Azure Functions Development

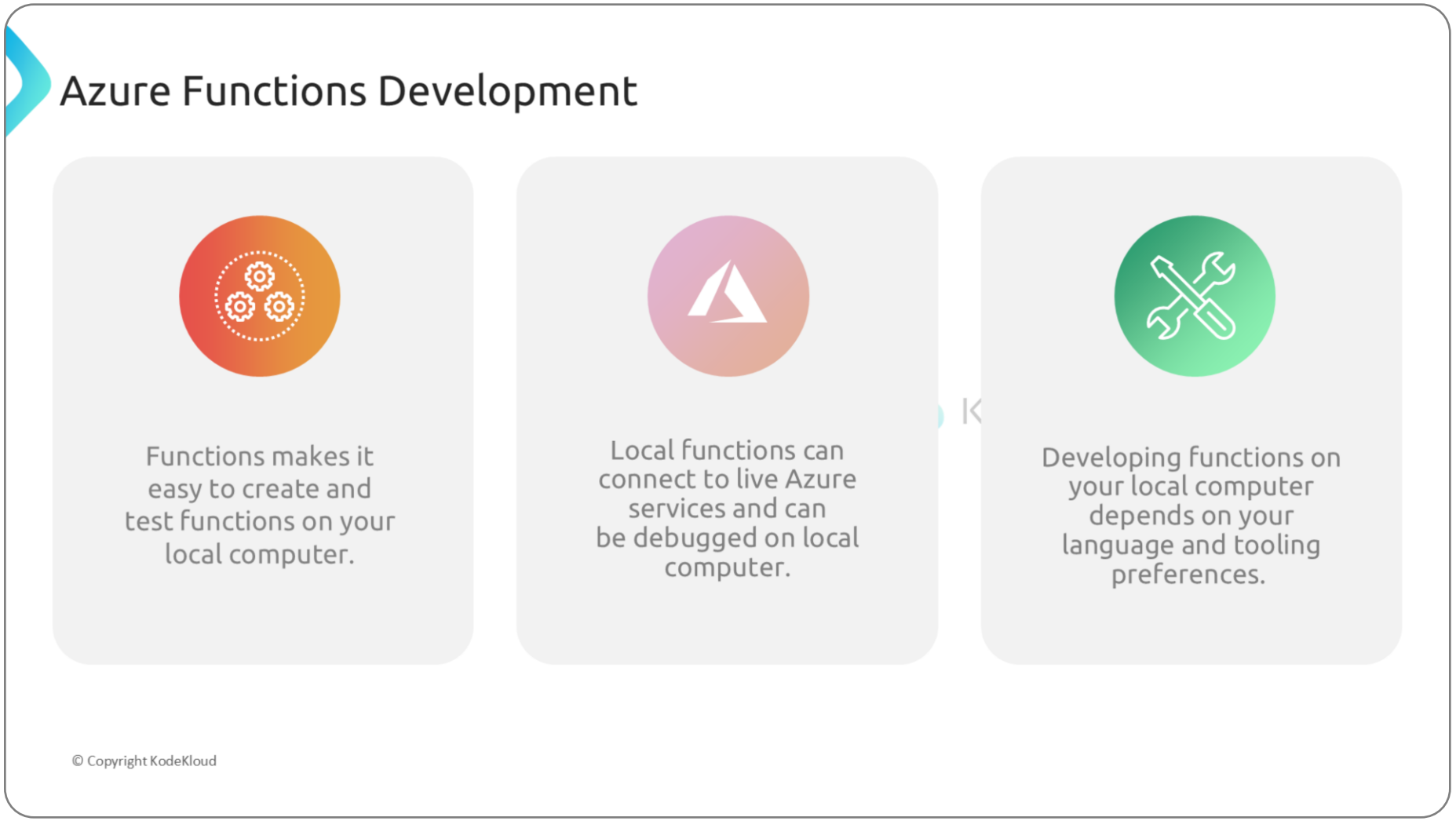
It is a unit of deployment and management for your functions.

A function app comprises one or more individual functions that are managed, deployed, and scaled together.

All functions in a function app share the same pricing plan, deployment method, and runtime version.

A function app provides an execution context in Azure to run your functions.

-It is the unit of deployment and management for your functions.
-A function app is comprised of one or more individual functions that are managed, deployed, and scaled together.
-All functions in a function app share the same pricing plan, deployment method, and runtime version.



WARNING: Do not mix local development with portal development in the same function app. When you create and publish functions from a local project, you should not try to maintain or modify project code in the portal.

Local development environments

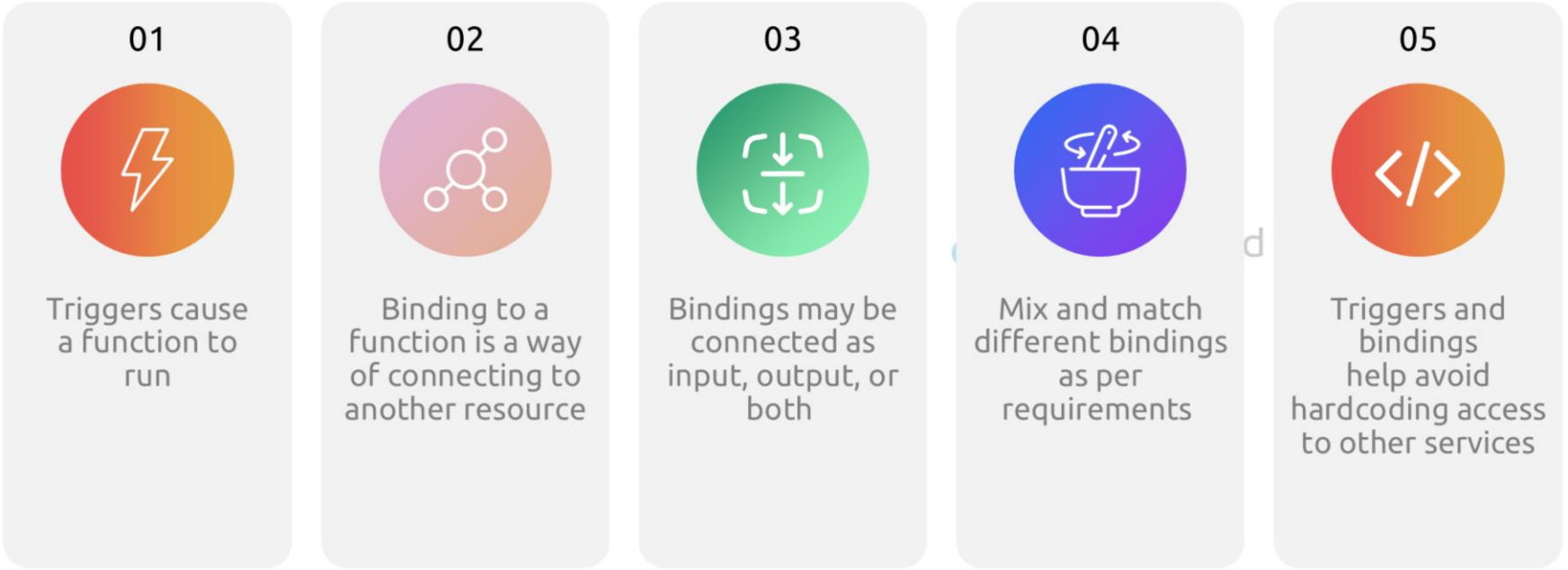
Functions makes it easy to use your favorite code editor and development tools to create and test functions on your local computer.

Your local functions can connect to live Azure services, and you can debug them on your local computer using the full

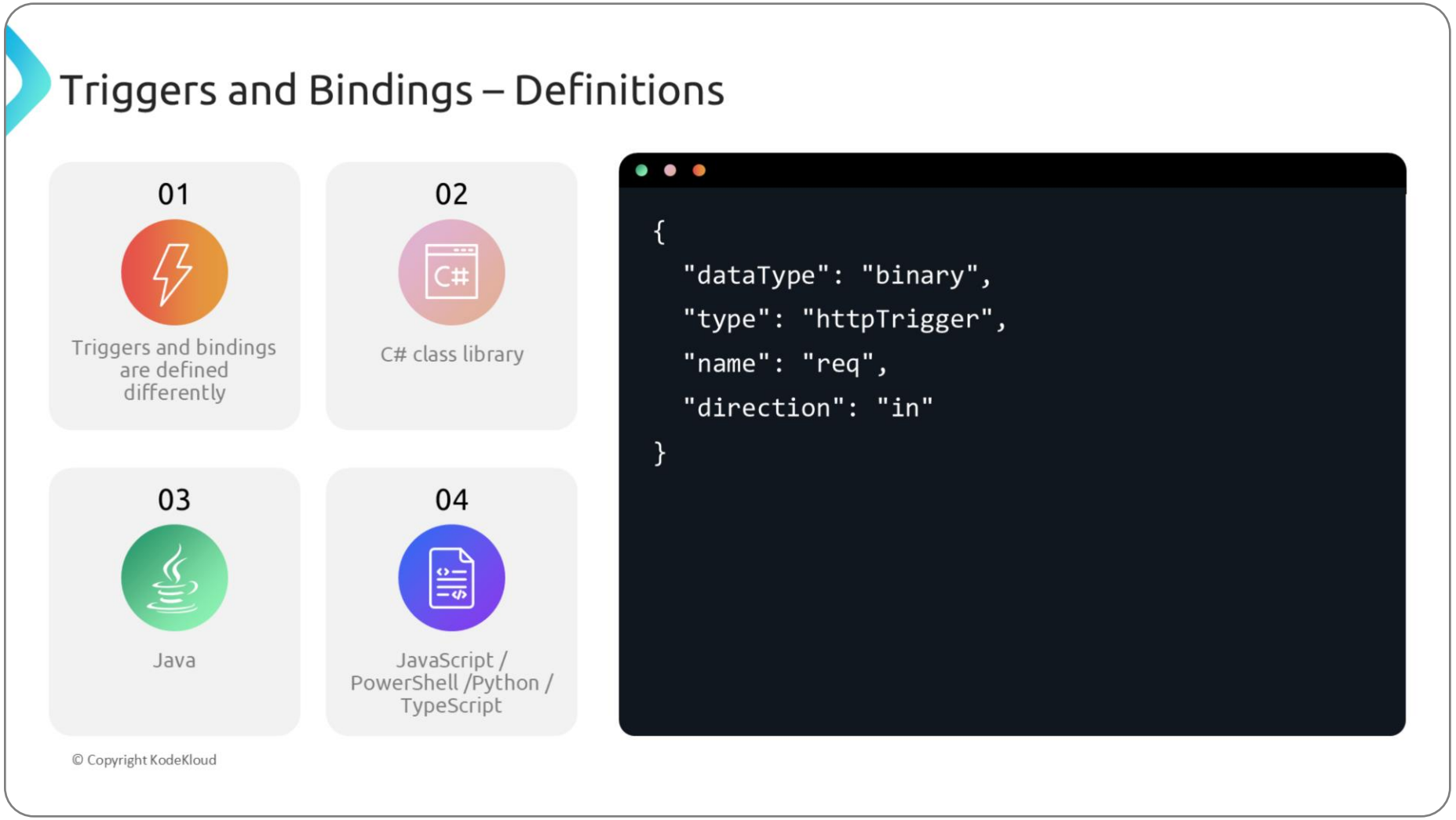
Functions runtime.
The way in which you develop functions on your local computer depends on your language and tooling preferences.

Triggers and Bindings

Creating Triggers and Bindings



Triggers are what cause a function to run. A trigger defines how a function is invoked and a function must have exactly one trigger.
Binding to a function is a way of declaratively connecting another resource to the function
Bindings may be connected as input bindings, output bindings, or both.
You can mix and match different bindings to suit your needs.
Triggers and bindings let you avoid hardcoding access to other services.



In .NET and Java, the parameter type defines the data type for input data. For instance, use string to bind to the text of a queue trigger, a byte array to read as binary, and a custom type to de-serialize to an object. Since .NET class library functions and Java functions don't rely on function.json for binding definitions, they can't be created and edited in the portal. C# portal editing is based on C# script, which uses function.json instead of attributes.

For languages that are dynamically typed such as JavaScript, use the dataType property in the function.json file. For example, to read the content of an HTTP request in binary format, set dataType to binary.

Triggers and bindings are defined differently depending on the development language.
C# class library - decorating methods and parameters with C# attributes
Java - decorating methods and parameters with Java annotations
JavaScript/PowerShell/Python/TypeScript - updating function.json schema

Binding Direction

01

All triggers and bindings have a direction property in the function.json file.

02

For triggers, the direction is always in.

03

Input and output bindings use in and out.

04

Some bindings support a special direction inout. Only the Advanced editor is available in this.

05

When you use attributes in a class library, the direction is provided in an attribute constructor or inferred from parameter type.

Azure Functions – Triggers and Bindings Example

Scenario: You want to write a new row to Azure Table storage whenever a new message appears in Azure Queue storage.

This scenario can be implemented using an Azure Queue storage trigger and an Azure Table storage output binding.

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


The first element in the bindings array is the Queue storage trigger. The type and direction properties identify the trigger. The name property identifies the function parameter that receives the queue message content. The name of the queue to monitor is in queueName, and the connection string is in the app setting identified by connection.

The second element in the bindings array is the Azure Table Storage output binding. The type and direction properties identify the binding. The name property specifies how the function provides the new table row, in this case by using the function return value. The name of the table is in tableName, and the connection string is in the app setting identified by

connection.

C# Script Example

```
...
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; }
}
```


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

C# script code that works with the previous trigger and binding specified.

JavaScript Example

```
module.exports = async function (context, order) {
    order.PartitionKey = "Orders";
    order.RowKey = generateRandomId();
    context.bindings.order = order;
};
function generateRandomId() {
    return Math.random().toString(36).substring(2, 15) +
        Math.random().toString(36).substring(2, 15);
}
```


From an incoming queue message that is a JSON object, add fields and write to Table Storage
The second parameter to context.done is used as the value for the new row

JavaScript code that works with the previous trigger and binding specified.

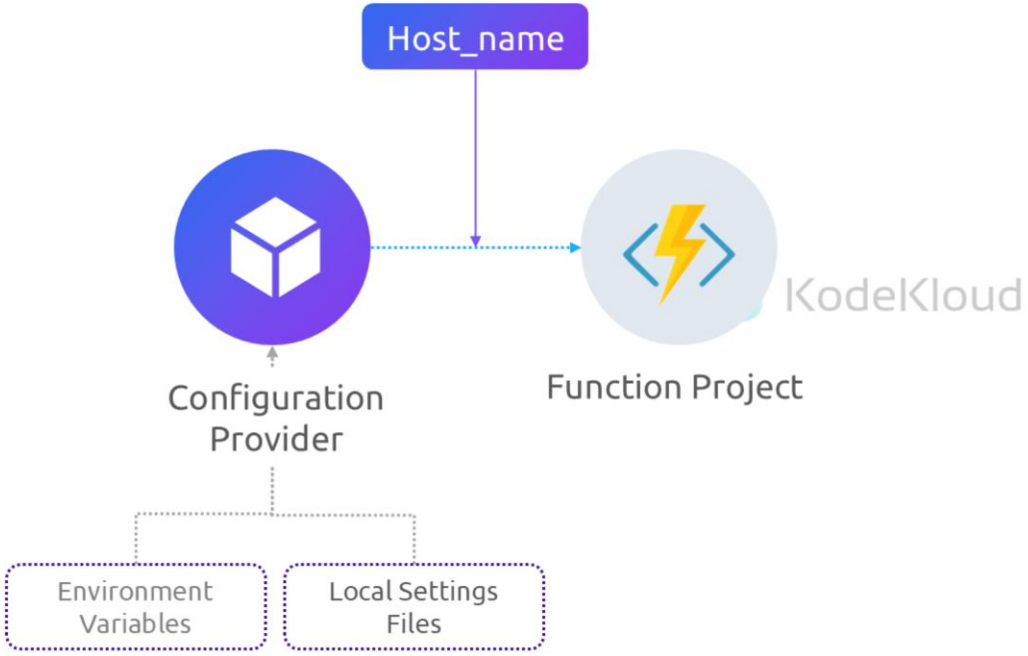
Class Library Example

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

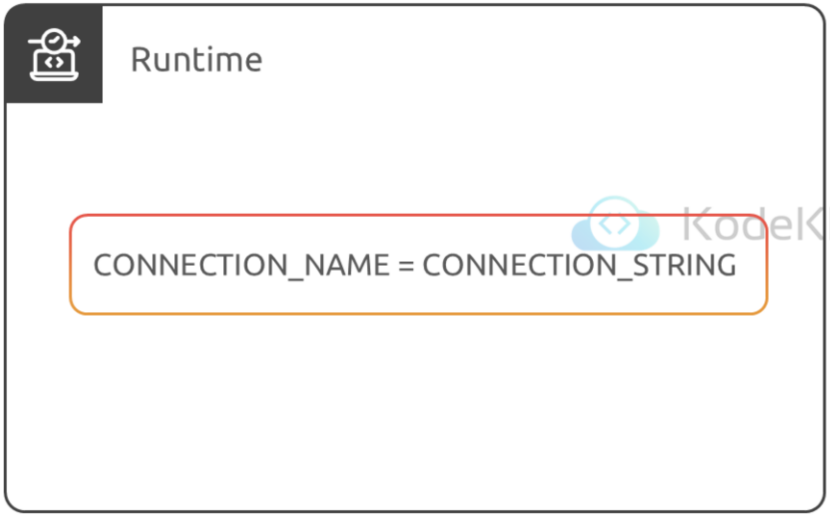

In a class library, the same trigger and binding information — queue and table names, storage accounts, function parameters for input and output — is provided by attributes instead of a function.json file.

Connecting to Azure Services

Connecting Functions to Azure Services

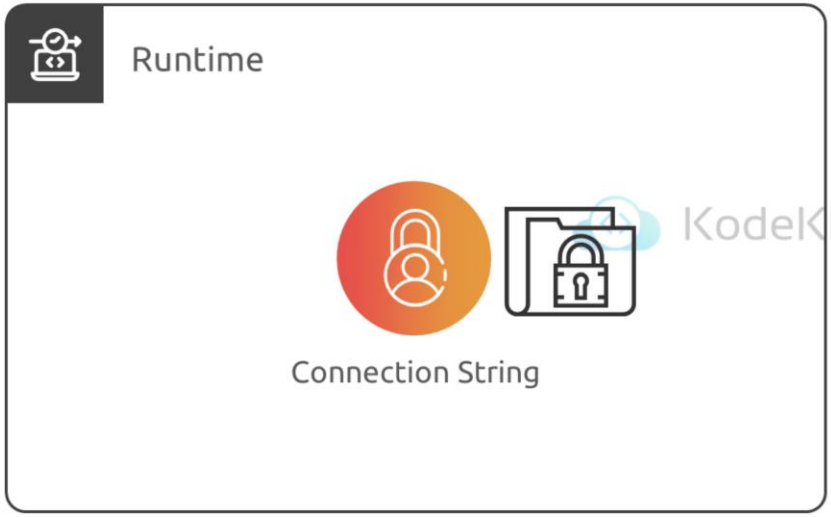


Connection Values



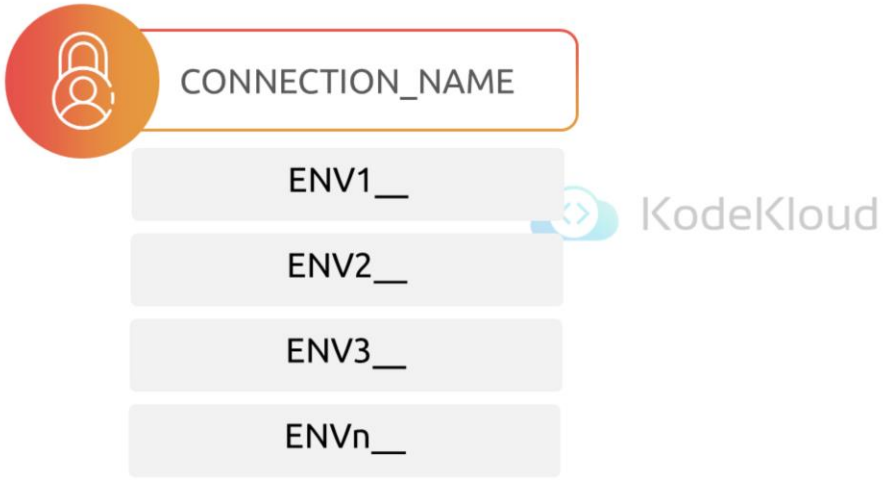
When the connection name resolves to a single exact value, the runtime identifies the value as a connection string, which typically includes a secret.

Connection Values



When the connection name resolves to a single exact value, the runtime identifies the value as a connection string, which typically includes a secret.

Connection Values



A connection name can also refer to a collection of multiple configuration items.
Environment variables can be treated as a collection by using a shared prefix that ends in double underscores __.

Connecting Functions to Azure Services

Configure an identity-based connection

- Some connections in Azure Functions are configured to use an identity instead of a secret.
- In some cases, a connection string may still be required.

Grant permission to the Identity

- Always use permissions to perform the intended actions.
- Typically done by assigning a role in Azure RBAC or specifying the identity in an access policy.

Configure an identity-based connection
Note: Identity-based connections are not supported with Durable Functions.
When hosted in the Azure Functions service, identity-based connections use a managed identity. The system-assigned identity is used by default, although a user-assigned identity can be specified with the credential and clientID properties. When run in other contexts, such as local development, your developer identity is used instead, although this can be customized using alternative connection parameters.

Grant permission to the identity
Important: Some permissions might be exposed by the target service that are not necessary for all contexts. Where possible, adhere to the principle of least privilege, granting the identity only required privileges.

KodeKloud

Visit www.kodekloud.com to learn more.