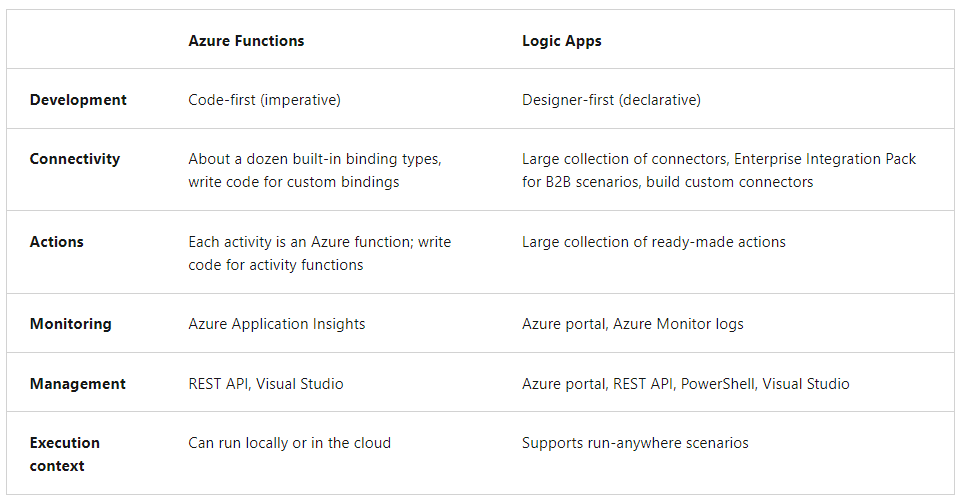
**Discover Azure Functions**

* Azure Functions are a great solution for processing data, integrating systems, working with the internet-of-things (IoT), and building simple APIs and microservices.
* Azure Functions supports triggers, which are ways to start execution of your code, and bindings, which are ways to simplify coding for input and output data.

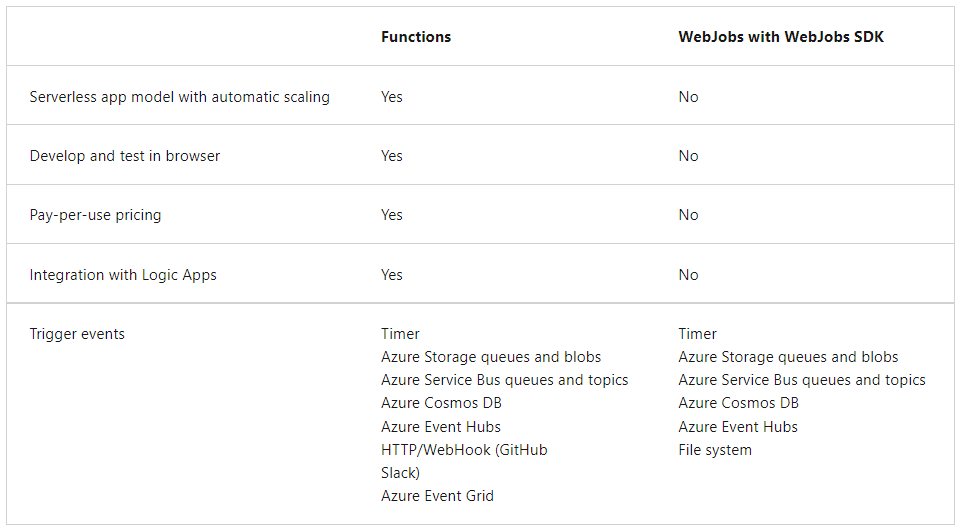
**Compare Azure Functions and Azure Logic Apps**

* Both Functions and Logic Apps enable serverless workloads.
* Azure Functions is a serverless compute service, whereas Azure Logic Apps provides serverless workflows.
* 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.

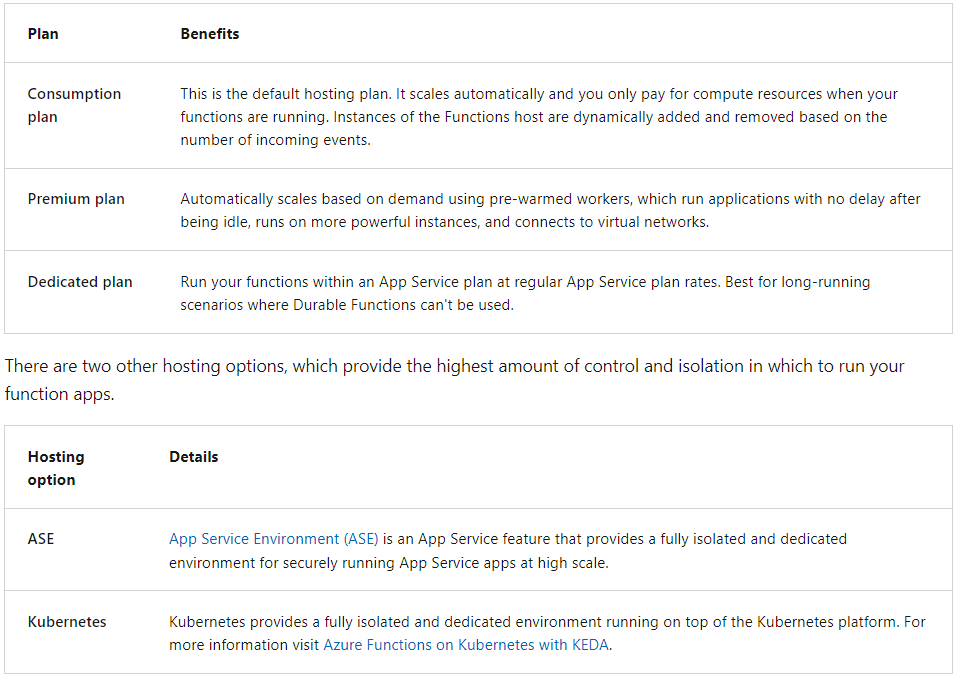


**Compare Functions and WebJobs**

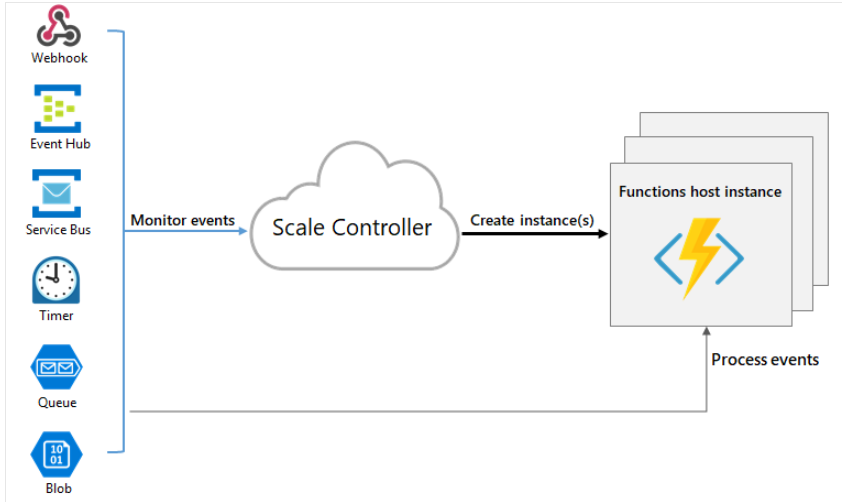
* Like Azure Functions, Azure App Service WebJobs with the WebJobs SDK is a code-first integration service that is designed for developers.
* Both are built on Azure App Service and support features such as source control integration, authentication, and monitoring with Application Insights integration.



**Compare Azure Functions hosting options**

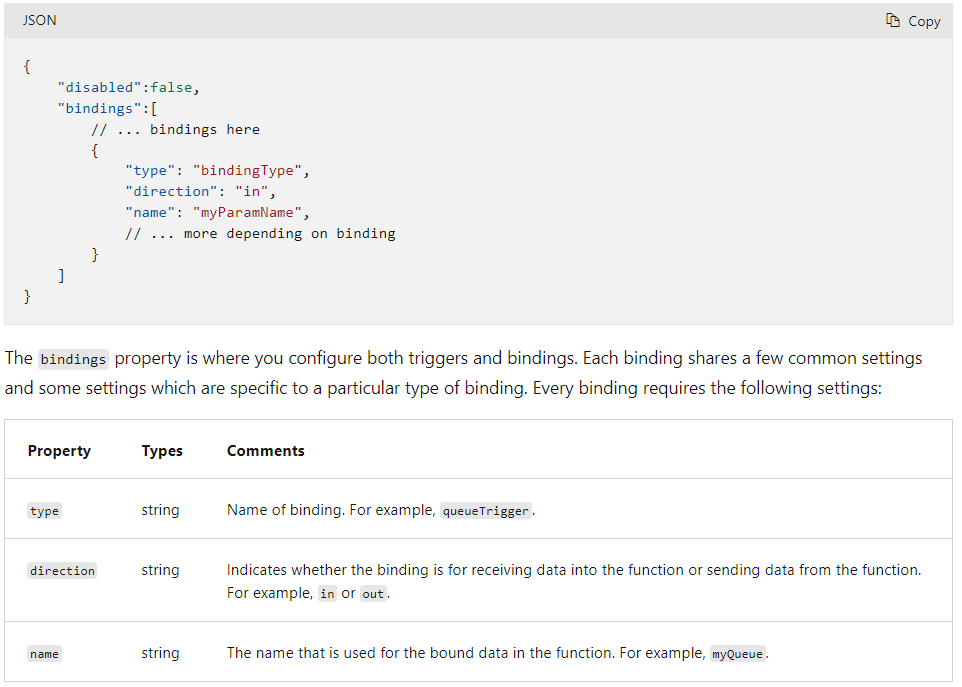


**Scale Azure Functions**

* Each instance of the Functions host in the Consumption plan is limited to 1.5 GB of memory and one CPU.
* Function code files are stored on Azure Files shares on the function's main storage account.
* When you delete the main storage account of the function app, the function code files are deleted and cannot be recovered.
* Azure Functions uses a component called the **scale controller** to monitor the rate of events and determine whether to scale out or scale in.
* The scale controller uses heuristics for each trigger type. For example, when you're using an Azure Queue storage trigger, it scales based on the queue length and the age of the oldest queue message.  
    
  
* **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**
  + 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
  + 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.
* Using an App Service plan, you can manually scale out by adding more VM instances. You can also enable autoscale, though autoscale will be slower than the elastic scale of the Premium plan.

**A function contains two important pieces**

* Code (Your code, which can be written in a variety of languages) and function.json
* **function.json**
  + For compiled languages, this config file is generated automatically from annotations in your code.
  + For scripting languages, you must provide the config file yourself.
  + **The function.json file defines the** 
    1. function's trigger
    2. bindings
    3. other configuration settings.
  + **Every function has one and only one trigger.**
  + **Bindings are optional and a function might have one or multiple input and/or output bindings.**
  + The runtime uses this config file to determine the events to monitor and how to pass data into and return data from a function execution.
* The following is an example function.json file.



**Function app**

* A function app provides an execution context in Azure in which your functions run.
* As such, 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 of the functions in a function app share the same pricing plan, deployment method, and runtime version.
* Think of a function app as a way to organize and collectively manage your functions.

Note: In Functions 2.x all functions in a function app must be authored in the same language. In previous versions of the Azure Functions runtime, this wasn't required.

* Folder Structure
  + The **host.json** file contains runtime-specific configurations and is in the root folder of the function app.
  + A bin folder contains packages and other library files that the function app requires.
  + Specific folder structures required by the function app depend on language:

**Triggers and Bindings**

* Triggers
  + Triggers are what cause a function to run.
  + A trigger defines how a function is invoked and a function must have exactly one trigger.
  + Triggers have associated data, which is often provided as the payload of the function.
* Bindings
  + 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.
  + Data from bindings is provided to the function as parameters.
  + You can mix and match different bindings to suit your needs.
  + Bindings are optional and a function might have one or multiple input and/or output bindings.
* For languages that rely on function.json, the portal provides a UI for adding bindings in the Integration tab. You can also edit the file directly in the portal in the Code + test tab of your function.
* 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.

**Binding direction**

* All triggers and bindings have a direction property in the function.json file:
* For triggers, the direction is always in
* Input and output bindings use in and out
* Some bindings support a special direction inout. If you use inout, only the Advanced editor is available via the Integrate tab in the portal.
* When you use attributes in a class library to configure triggers and bindings, the direction is provided in an attribute constructor or inferred from the parameter type

**Connection values**

* The runtime identifies the value as a connection string, which typically includes a secret.
* Some connections in Azure Functions are configured to use an identity instead of a secret.
* 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.

**Durable Functions**

* The durable functions extension lets you define stateful workflows by writing orchestrator functions and stateful entities by writing entity functions using the Azure Functions programming model.
* Behind the scenes, the extension manages state, checkpoints, and restarts for you, allowing you to focus on your business logic.
* You can use Durable Functions to create flexible recurrence intervals, manage task lifetimes, and create multiple monitor processes from a single orchestration.

**Application patterns**

* The primary use case for Durable Functions is simplifying complex, stateful coordination requirements in serverless applications.
* The following sections describe typical application patterns that can benefit from Durable Functions:
  + **Function chaining**
    - In the function chaining pattern, a sequence of functions executes in a specific order.
    - In this pattern, the output of one function is applied to the input of another function.
  + **Fan-out/fan-in**
    - In the fan out/fan in pattern, you execute multiple functions in parallel and then wait for all functions to finish.
    - Often, some aggregation work is done on the results that are returned from the functions.
    - The work is tracked by using a dynamic list of tasks. The .NET Task.WhenAll API or JavaScript context.df.Task.all API is called, to wait for all the called functions to finish.
  + **Async HTTP APIs**
    - The async HTTP API pattern addresses the problem of coordinating the state of long-running operations with external clients.
    - Durable Functions provides built-in support for this Async HTTP APIs pattern
  + Monitor
  + Human interaction

**Four function types**

1. Orchestrator Functions

* Orchestrator functions describe how actions are executed and the order in which actions are executed.
* Orchestration instances are single-threaded so it isn't necessary to worry about race conditions within an orchestration.
* Orchestrator functions aren't permitted to do I/O.
* Orchestrator functions have the ability to wait and listen for external events.
* The WaitForExternalEvent (.NET), waitForExternalEvent (JavaScript), and wait\_for\_external\_event (Python) methods of the orchestration trigger binding allows an orchestrator function to asynchronously wait and listen for an external event.
* The RaiseEventAsync (.NET) or raiseEvent (JavaScript) method of the orchestration client binding sends the events that WaitForExternalEvent (.NET) or waitForExternalEvent (JavaScript) waits for. The RaiseEventAsync method takes eventName and eventData as parameters. The event data must be JSON-serializable.

1. Activity Functions
   * + - Activity functions are the basic unit of work in a durable function orchestration.
       - For example, you might create an orchestrator function to process an order. The tasks involve checking the inventory, charging the customer, and creating a shipment. Each task would be a separate activity function.
       - These activity functions may be executed serially, in parallel, or some combination of both.
       - Activity functions are frequently used to make network calls or run CPU intensive operations.
       - An activity function can also return data back to the orchestrator function.
       - **It isn't possible to pass multiple parameters to an activity function directly.**
       - **The recommendation is to pass in an array of objects or to use ValueTuples objects in .NET**.
2. Entity Functions
   * + - Entity functions define operations for reading and updating small pieces of state. We often refer to these stateful entities as durable entities.
3. Client Functions
   * + - Unlike other function types, orchestrator and entity functions cannot be triggered directly using the buttons in the Azure portal.
       - If you want to test an orchestrator or entity function in the Azure portal, you must instead run a client function that starts an orchestrator or entity function as part of its implementation.

**Task Hub**

* A task hub in Durable Functions is a logical container for durable storage resources that are used for orchestrations and entities.
* Orchestrator, activity, and entity functions can only directly interact with each other when they belong to the same task hub.
* If multiple function apps share a storage account, each function app must be configured with a separate task hub name (in host.json file).
* A storage account can contain multiple task hubs.
* This restriction generally applies to other storage providers as well.
* **A task hub in Azure Storage consists of the following resources**:
  + One or more control queues.
  + One work-item queue.
  + One history table.
  + One instances table.
  + One storage container containing one or more lease blobs.
  + A storage container containing large message payloads, if applicable.
* **Task hub names rules**
  + Contains only alphanumeric characters
  + Starts with a letter
  + Has a minimum length of 3 characters, maximum length of 45 characters