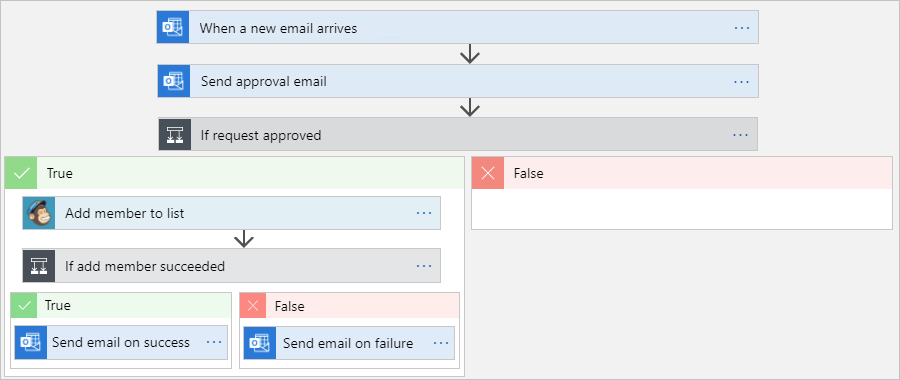
Reference Architecture 1

----------------------------------------

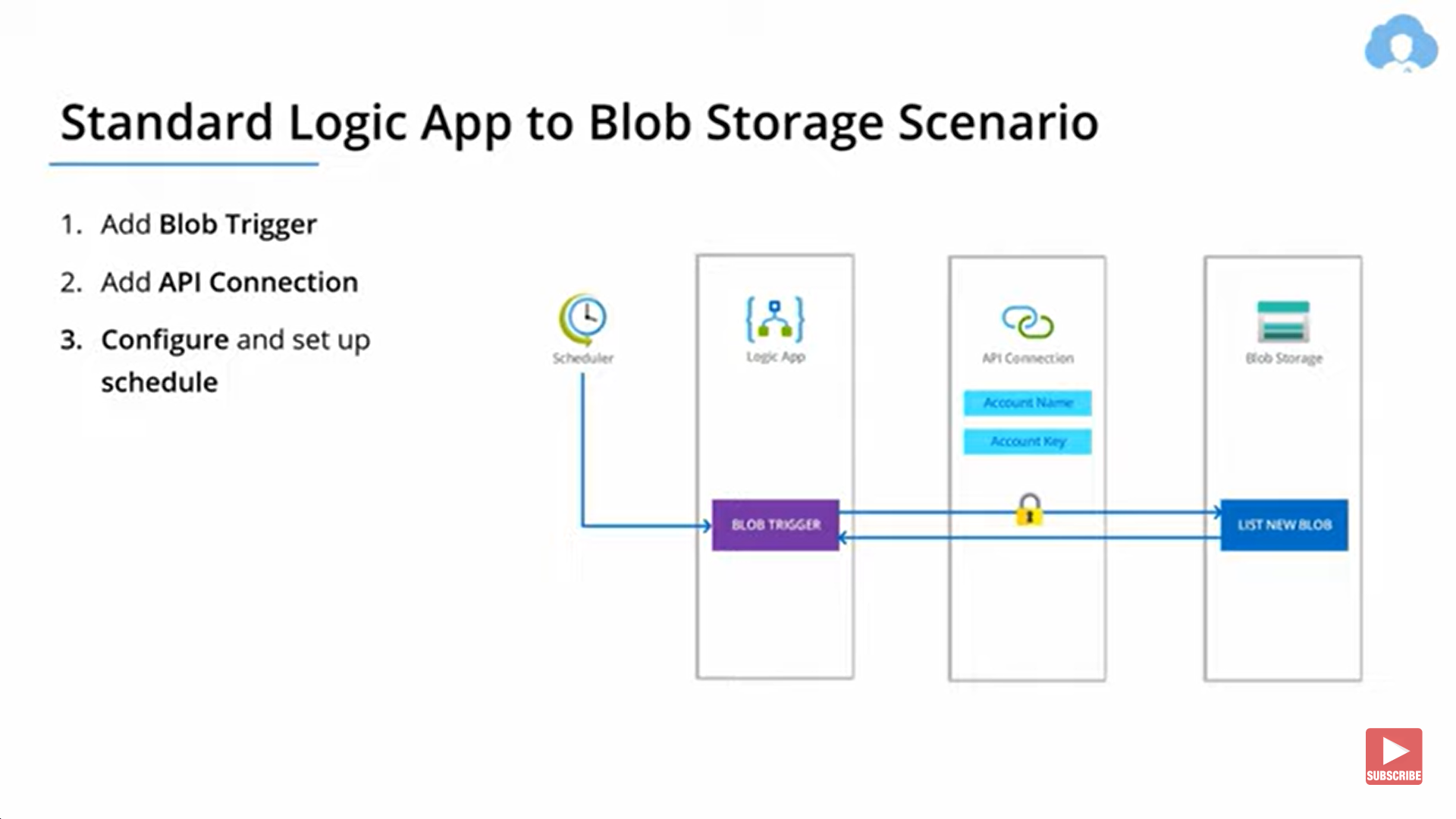
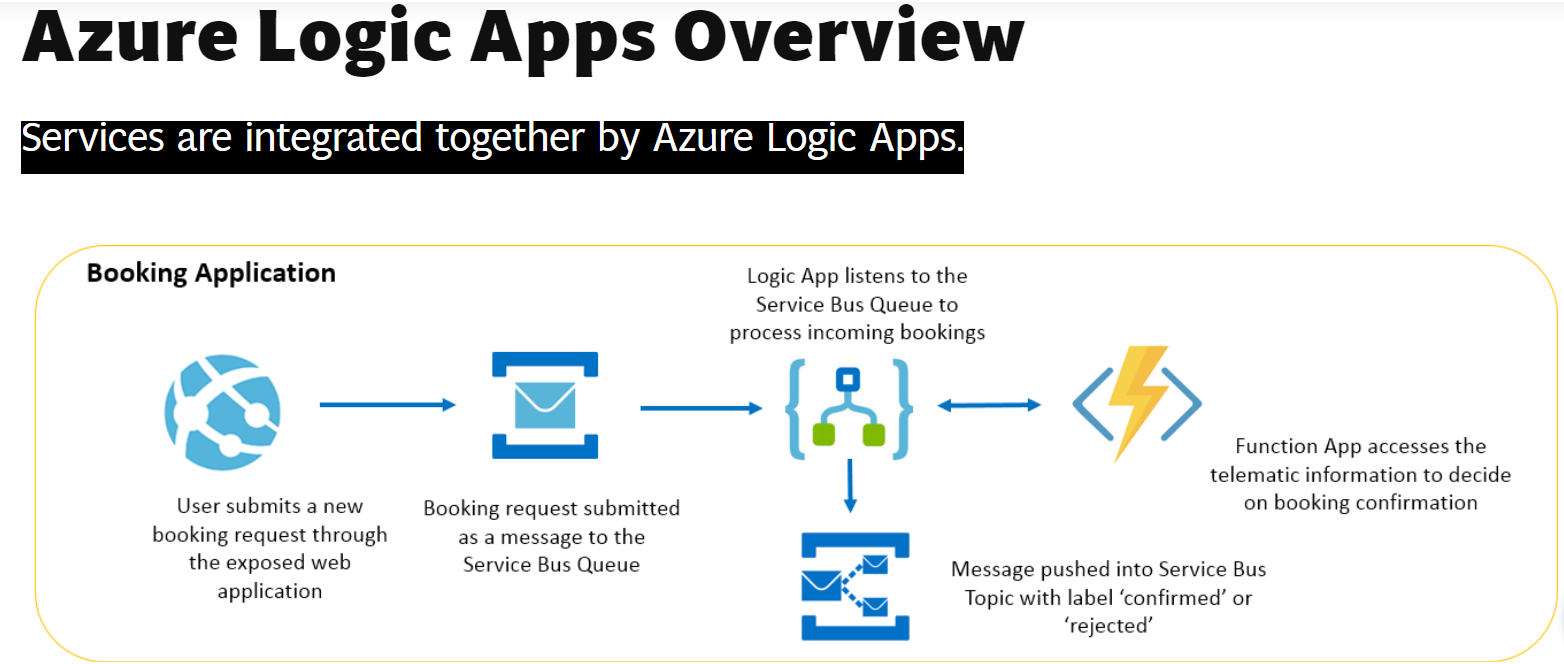
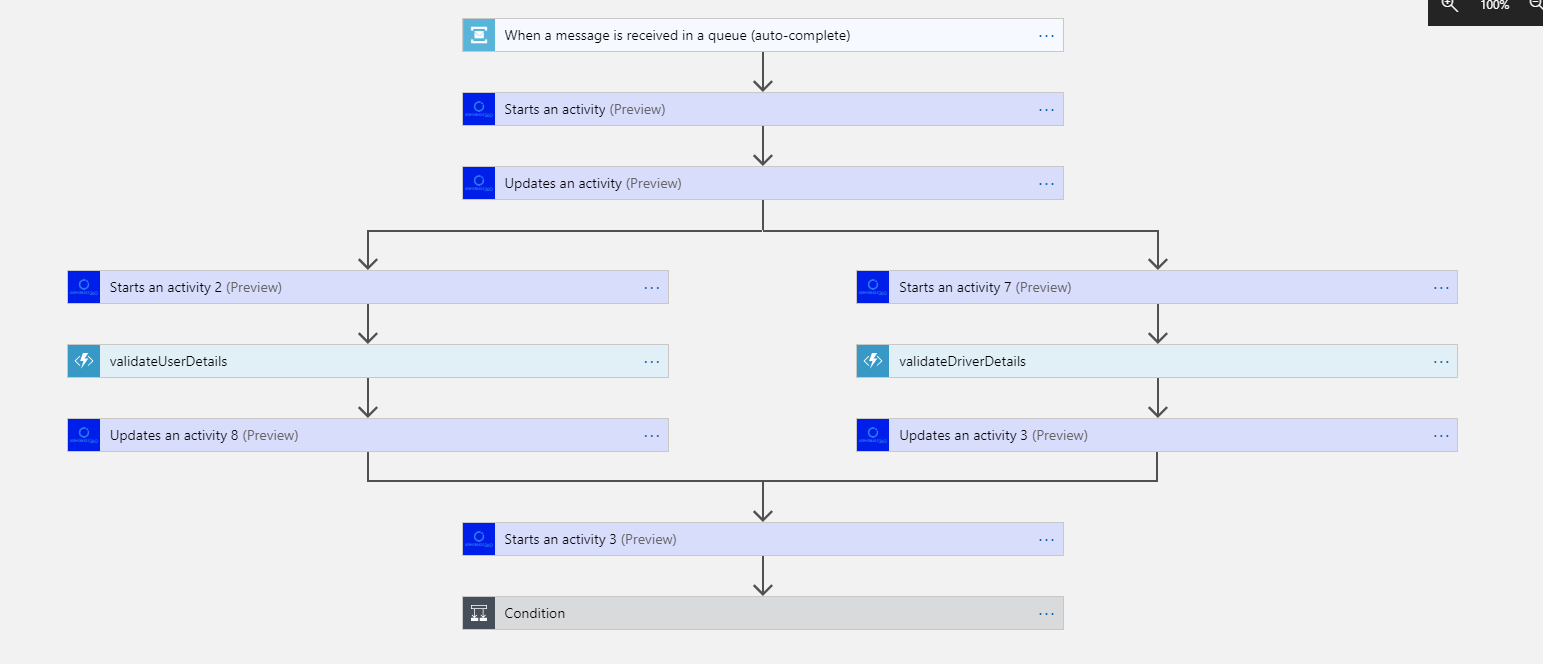
<https://learn.microsoft.com/en-us/azure/logic-apps/tutorial-process-mailing-list-subscriptions-workflow>

<https://learn.microsoft.com/en-us/azure/logic-apps/tutorial-process-mailing-list-subscriptions-workflow>

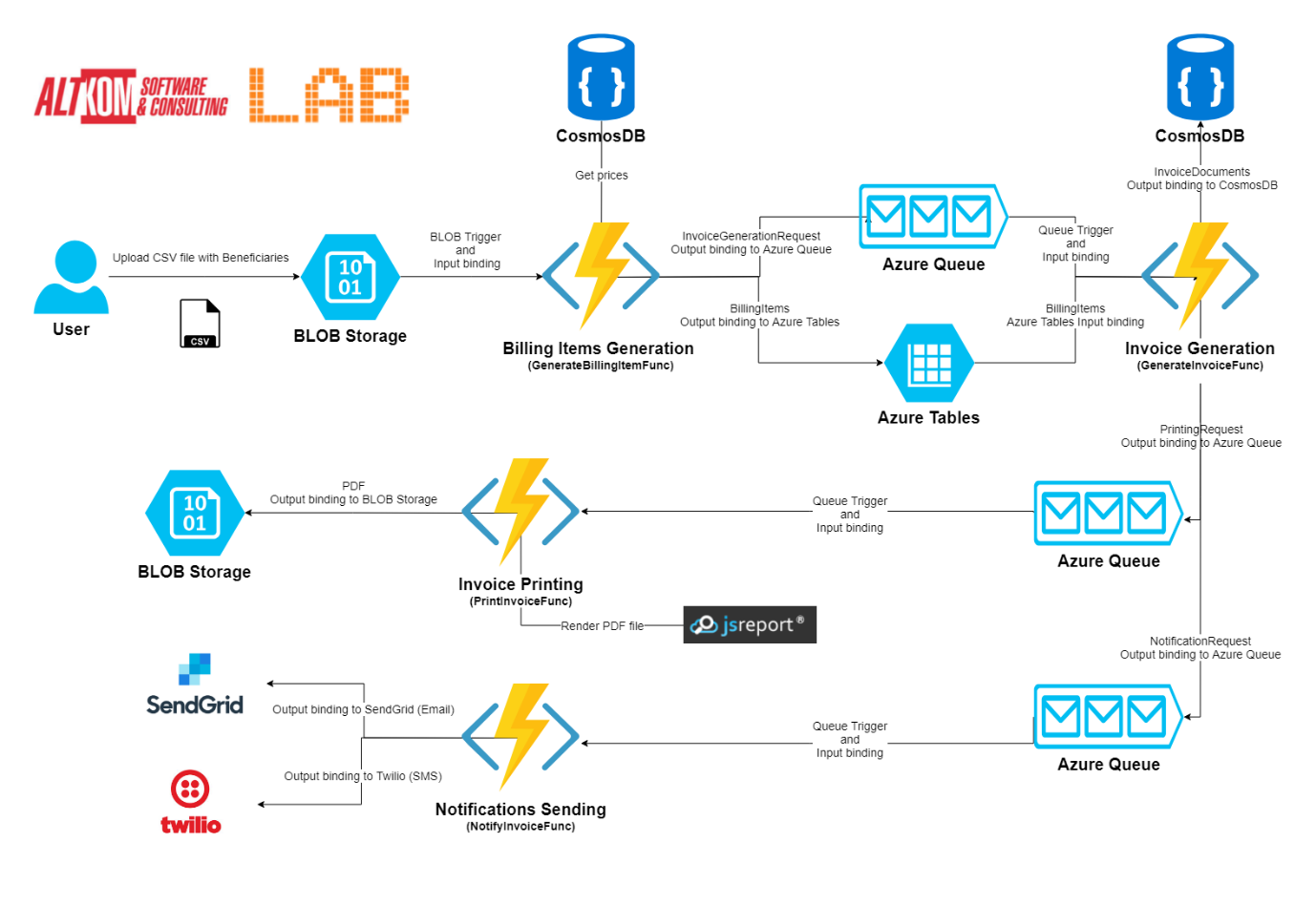


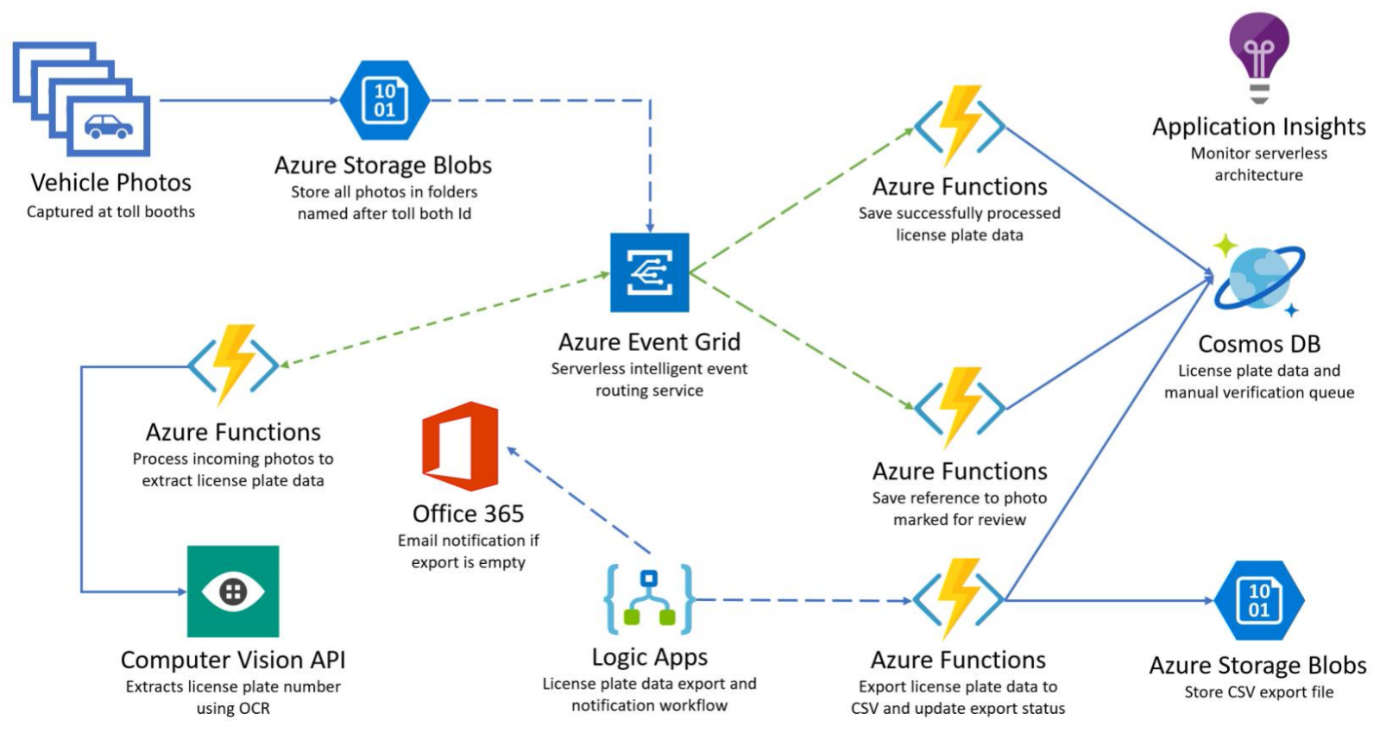
|  |  |
| --- | --- |
| **Azure Logic App**  Common issue   * Exception handling * Timeout   Solution |  |
|  | <https://devpress.csdn.net/cloudnative/62f2c9777e66823466185a9a.html> |
|  | Enable Managed identity: |
|  | * Create a blank logic app. * Add a trigger that monitors emails for subscription requests. * Add an action that sends emails for approving or rejecting these requests. * Add a condition that checks the approval response. * Add an action that adds approved members to the mailing list. * Add a condition that checks whether these members successfully joined the list. * Add an action that sends emails confirming whether these members successfully joined the list.   High-level finished logic app overview |
|  |  |
|  |  |
|  |  |
|  |  |

Azure Logic App

*  
* 

Azure function:





[FunctionName("MyFirstFunction")]

public static void QueueTrigger(

[QueueTrigger("myqueue-items")] string myQueueItem,

ILogger log)

{

log.LogInformation($"C# function processed: {myQueueItem}");

}

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  | [FunctionName("BookConference")]  public static ConfTicket BookConference([ActivityTrigger] string conference)  {  var ticket = BookingService.Book(conference);  return new ConfTicket { Code = ticket };  } |
| Error-handling and retries | [FunctionName("SequentialWorkflow")]  public static async Task Sequential([OrchestrationTrigger] DurableOrchestrationContext context)  {  var conference = await context.CallActivityAsync<ConfTicket>("BookConference", "ServerlessDays");  var flight = await context.CallActivityAsync<FlightTickets>("BookFlight", conference.Dates);  await context.CallActivityAsync("BookHotel", flight.Dates);  } |
|  | [FunctionName("CombinedOrchestrator")]  public static async Task CombinedOrchestrator([OrchestrationTrigger] DurableOrchestrationContext context)  {  await context.CallSubOrchestratorAsync("BookTrip", serverlessDaysAmsterdam);  await context.CallSubOrchestratorAsync("BookTrip", serverlessDaysHamburg);  } |
|  | [FunctionName("ParallelWorkflow")]  public static async Task Parallel([OrchestrationTrigger] DurableOrchestrationContext context)  {  var amsterdam = context.CallSubOrchestratorAsync("BookTrip", serverlessDaysAmsterdam);  var hamburg = context.CallSubOrchestratorAsync("BookTrip", serverlessDaysHamburg);  var expenses = await Task.WhenAll(amsterdam, hamburg);  await context.CallActivityAsync("ReportExpenses", expenses);  } |
| **Fan-out / Fan-in** | var emailSendingTasks =  recepients  .Select(to => context.CallActivityAsync<bool>("SendEmail", to))  .ToArray();  var results = await Task.WhenAll(emailSendingTasks);  if (results.All(r => r)) { /\* ... \*/ } |
| 1. Function chaining 2. Fan-out/Fan-in 3. Async HTTP APIs 4. Monitoring 5. Human interaction 6. Aggregator |  |

**Durable Functions** is an extension of Azure Functions through which you can define **a stateful workflow using one or more orchestrator functions**. You can focus on business logic without worrying about managing states and checkpoints.

With Durable Functions, you use different types of functions that work together to reach a common goal. Each function is intended to play a different role in the workflow.

* **Orchestrator Function**. As the name suggests, this type of function orchestrates the workflow. Here you decide how and when other functions are being called. Orchestrator functions have to be ***deterministic*** which means that the result can never change between subsequent executions, and therefore actions you can perform in this type of function are subject to constraints.
* **Activity Function.**You can think of an activity function as a ***worker*** invoked by the orchestrator that takes charge of a task and performs some activities in order to complete it.
* **Entity Function.**It provides a way to read and update small ***states***. An entity has a unique identity and an internal state. You can refer to the entity through its identifier and read and update its state.
* **Client Function.**This is the ***entry point*** of your workflow. It is necessary to trigger an orchestrator or entity function. A client function uses the *durable client output binding*and can be triggered by all the available triggers as for classic Azure Functions.



|  |
| --- |
| [FunctionName(nameof(UserMonitorWorkflow))] |
|  | public static async Task UserMonitorWorkflow( |
|  | [OrchestrationTrigger]DurableOrchestrationContext context, |
|  | ILogger logger) |
|  | { |
|  | var username = context.GetInput<string>(); |
|  | var expiryTime = context.CurrentUtcDateTime.AddHours(Global.MonitorTimeoutHours); |
|  | var timeout = false; |
|  | while (context.CurrentUtcDateTime < expiryTime) |
|  | { |
|  | timeout = false; |
|  | var done = await context.CallActivityAsync<bool>(nameof(MonitorUser), username); |
|  | if (done) |
|  | { |
|  | break; |
|  | } |
|  | var nextCheck = context.CurrentUtcDateTime.AddSeconds(CheckIntervalSeconds); |
|  | await context.CreateTimer(nextCheck, CancellationToken.None); |
|  | timeout = true; |
|  | } |
|  | if (timeout) |
|  | { |
|  | // timed out in 1 hour |
|  | } |
|  | // ended due to user dying or winning the game |
|  | } |