**Resources:**

1. [Azure Functions documentation | Microsoft Learn](https://learn.microsoft.com/en-us/azure/azure-functions/) - root of MS's documentation on AF, which is pretty good
2. [Azure Functions Overview | Microsoft Learn](https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview?pivots=programming-language-csharp)
3. [Create a C# function using Visual Studio Code - Azure Functions | Microsoft Learn](https://learn.microsoft.com/en-us/azure/azure-functions/create-first-function-vs-code-csharp)
4. [Azure Functions scale and hosting | Microsoft Learn](https://learn.microsoft.com/en-us/azure/azure-functions/functions-scale)
5. [Function types in Azure Durable Functions | Microsoft Learn](https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-types-features-overview)

What are Azure Functions

Azure Functions, created in 2016 by MS, are a serverless compute service that allows you to run event-triggered small pieces of code, or "functions," in the cloud - without having to explicitly provision or manage infrastructure. This means you can focus on writing the code that matters most to you, while Azure handles the rest. Azure Functions can be triggered by various events, such as HTTP requests, timers, or messages from other Azure services, making them highly versatile for different use cases.

**Benefits of Azure Functions**

1. **Serverless Architecture**: Azure Functions is a serverless compute service, meaning **you don't have to manage infrastructure (no Server Provisioning or maintenance: No need to set up or maintain physical or virtual servers)**. The service **automatically scales based on demand, and you only pay for the time your code runs**. Azure Functions automatically scale based on demand, but it's important to choose the correct hosting plan to ensure optimal performance and cost-efficiency. - MACHINES ARE AUTOMATICALLY ADDED IF MORE MEMORY/CPU IS NEEDED
2. Have a lot of useful **build-in functionality**, like retry logic on failure.
3. **Event-Driven**: Functions can be triggered by **various events from Azure services or external services**. This makes them suitable for real-time data processing, such as handling HTTP requests, running tasks on a timer, or responding to changes in storage.
4. **Cost-Effective**: With the different consumption plans, you only pay for the compute resources when your functions are running. This can lead to significant cost savings, especially for **applications with variable workloads**.
5. **Language Support**: Azure Functions **supports multiple programming languages**, including C#, Java, JavaScript, PowerShell, and Python. This flexibility allows developers to use their preferred language. - TALK ABOUT MIXING LANGUAGES
6. **Integration with Azure Services**: Azure Functions integrates seamlessly with other Azure services like Azure Storage, Azure Cosmos DB, and Azure Event Hubs. No writing custom logic for auth or connecting to Azure service. This makes it easier to build complex workflows and applications.

**Limitations of Azure Functions**

1. **Timeout Restrictions**: The default execution timeout for HTTP-triggered functions on the Consumption plan is 5 minutes, which can be extended to a maximum of 10 minutes. This can be a limitation for long-running tasks. - THERE ARE WAYS TO EXTEND TIME WITH MORE EXPENSIVE PLANS.
2. **Cold Start Latency**: Functions on the Consumption plan may experience cold start latency, which is the delay that occurs when a function is triggered after being idle. This can **impact performance for time-sensitive applications**. - COLD START REQUIRES TO ISNTALL ALL THE PREREQUEISITS FOR YOUR CODE TO RUN (KINDA STARTING A COMPUTER), SO THE MORE COMPLEX YOUR SOLUTION IS THE LONGER IT CAN BE; YOU CAN ALWAYS HAVE A NUMBER OF VMs READY, BUT THAT COSTS $.
3. **Resource Constraints**: Azure Functions **have memory and CPU limits**, which can affect the performance of resource-intensive tasks. Functions running on the same Function App share resources, so **careful planning** is needed to avoid resource contention. - WHICH IS A PROBLEM WE ARE RUNNING INTO DURING ONE OF CURRENT PROJECTS
4. **Limited Telemetry and Debugging**: Azure Functions lack some advanced telemetry and debugging features, which can make it challenging to diagnose issues in production environments. - AND WHAT RUNS LOCALLY DOES NOT NECESSERILY RUNS OK IN THE CLOUD (runs out memory, for example).
5. **Stateless by design:** Azure Functions are **stateless by design, meaning they do not maintain any state between executions**.

Azure Functions, introduced by Microsoft in 2016, revolutionized cloud computing by offering a serverless architecture that allowed developers to focus on code rather than infrastructure. However, traditional Azure Functions had limitations, particularly in managing long-running processes and maintaining state. To address these challenges, Microsoft introduced Durable Functions in 2017 as an extension of Azure Functions.

Durable Functions built on the robust foundation of Azure Functions, adding capabilities for **creating stateful workflows in a serverless environment**. This allowed to mitigate some of the stateless limitations of "old" Azure Functions.

**Complexity in Orchestration**: While Durable Functions provide orchestration capabilities, managing complex workflows can still be challenging and may require additional effort to ensure reliability and maintainability. - DEBUGGING AND LOGGING IS PARTICULARLY ANNOYING ☹

There are 4 types of Durable AF: Activity (stateless), Entity (stateful), Client (stateful), Orchestrator (stateful). Today we will be looking at stateless AF only.

Setting up for work

In Visual Studio Code, navigate to folder, clone the repo

A screen shot of a computer

Description automatically generated

Open that folder in VSC

Git status - Checkout new branch

1. Why do we want separate brunch - ask
2. Note that same steps can be done in VSC UI - or in power shell terminal

Creating first Azure Function

**Function App VS Azure Function Vs App Service Plan**

* **Azure Function:** A piece of logic that can be triggered by an event, such as an HTTP request or a file being added to Blob storage.
* **Function App:** The execution context in which Azure Functions run. It can contain multiple Azure Functions and is akin to a class containing methods.
* **App Service Plan:** A set of computational resources associated with a Function App. Different plans offer varying pricing, scaling, and resource allocation.

**Create a Hello World AF from template**

1. First, required extensions: links and instructions on these here [Develop Azure Functions by using Visual Studio Code | Microsoft Learn](https://learn.microsoft.com/en-us/azure/azure-functions/functions-develop-vs-code?tabs=node-v4%2Cpython-v2%2Cisolated-process%2Cquick-create&pivots=programming-language-csharp) (Azure Functions extension, Azure Tools lets you work with, well, Azure Resources)
2. F1 - AF new project > python > V2 > python env (virtual env if you have it) > select trigger HTTP

A screenshot of a computer program

Description automatically generated

1. Types of triggers - short list

A trigger is what causes a function to run. It defines how a function is invoked and every function must have exactly one trigger. Triggers can also pass data into your function, similar to how method calls work. Here are some common types of triggers:

* **HTTP Trigger**: This trigger allows your function to be invoked via HTTP requests. It's useful for creating APIs and responding to webhooks.
* **Timer Trigger**: This trigger allows your function to run on a schedule. It's useful for tasks that need to be performed at regular intervals, such as cleanup jobs or data processing.
* **Queue Trigger**: This trigger allows your function to process messages from an Azure Storage Queue. It's useful for decoupling components and handling asynchronous processing.
* **Blob Trigger**: This trigger allows your function to run when a new blob is created or an existing blob is updated in Azure Blob Storage. It's useful for processing files and images.
* **Event Hub Trigger**: This trigger allows your function to process events from Azure Event Hubs. It's useful for real-time data processing and analytics.

1. Auth - short description - pick Anonymous (reminder: auth is for deployed stuff, locally don't need it)
2. Note: automatically creates virtual environment > needs to install AF package (.\.venv\Scripts\activate, pip install azure-functions)
3. Look through the code, look at working example

**Deploy**

* Make sure you have Azure account
* These steps can also be done via the power shell terminal
* Requirements file
* F1 > AF: deploy
* Create new Function App - a lot of things happen automatically; if you want more control - create empty new Function App via Azure Portal and then deploy to it instead
* See new app in the Azure Tools > open in portal
* Get function URL - test it out (show there is an option to Test separately. But need to set up CORS policy)
* Who can see it? Everyone! Who can access it? Everyone! (Function Auth, remember?). Update to Function Auth, redeploy. Remove code, see 401. Function AUTH is your basic level of security for AF. But it's always better to layer multiple layers of security.
* In a full solution there are other ways to improve security. For example, restrict what IP addresses can call this endpoint, or hiding it behind the private network, etc. But this is not part of this discussion.  
  I want to emphasize is different Service Plans use different security features. We right now are on the cheapest one. Which is pay as you go - you only pay for the how long your functions run for and on how many machines. Security example of hammering the function that is wide open - you pay for it! Or even with code - your function still runs even if it needs to send Not Auth-d responses - you still pay for it.
* Look at resource group that was automatically created for us > why did we created storage account? It stores information related to tasks that AF executes so it knows what's next in the queue. (Not getting into details).
* Commit current changes.

Blob trigger function example

Let's create another function that would read data from Blob storage. Let's create new storage in Azure, and create new container in it.

1. Create another function
   1. Types of AF: activity, orchestrator, entity, client ([Function types in Azure Durable Functions | Microsoft Learn](https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-types-features-overview))
   2. Requirements for each type: stateless vs durable (stateful); this brings certain restrictions on the code for each type - that's why we need to read and put results into Blob storage and not in local storage - because next time your function runs, it might be on a different VM which does not have that file.
   3. F1 > AF - create new function, Blob trigger - RECOMMEND BLOB STORAGE WITH EVENT TRIGGER, but it requires a but extra set up and they have a bug on the template for it, but has a lot of similar things. So we would do just Blob trigger directly.  
        
      path is the name of the container, which is not obvious  
      use new storage

A screenshot of a computer

Description automatically generated

1. Created connection variable stored in local settings (right now) and contains all of the details of how to connect to that Blob storage
2. Notice, no HTTP endpoint when we run
3. Test by adding any file
4. Add code to trigger on a cvs file only and read that file (see the repo)
5. Read specific columns, add a new column with some values, and save the new file back into blob storage (see the repo)

Code is ready! Next:

1. Install required packages
2. Deploy - Oh no, we don't see the functions! Why?
3. Look in telemetry (app insights)  
   Explain environmental variables (local setting vs env variables when deployed)
4. Env variable for new storage needs to be added!

Talk about requirements file and that we need to update.

Make changes, deploy again.

Best Practices

When writing Activity Azure Functions, it’s important to follow best practices to ensure your functions are efficient, reliable, and maintainable. Here are some key recommendations:

1. **Keep Functions Small and Focused**: Each function should perform a single task or respond to a specific event. This makes them easier to test, debug, and maintain.
2. **Stateless Functions**: Design your functions to be stateless. This means they should not rely on any local state between executions. Use external storage solutions like Azure Storage or Azure SQL Database to maintain state if necessary.

**Key Principles of Stateless Functions**

* **No Local State**: Avoid storing any state locally within the function. This includes variables that persist between function executions. Instead, use external storage solutions like Azure Storage, Azure SQL Database, or Cosmos DB to maintain state. There are also type of AF called Entity Functions that can act as global variables between different instances.
* **Idempotency**: Design your functions to be idempotent, meaning they produce the same result even if executed multiple times. This is particularly important for functions triggered by events that might be reprocessed due to retries. Example or coding with idempotency in mind: Use “upsert” (update or insert) operations where possible. For example, instead of adding a new record, update an existing record if it already exists.  
  Random number generation, GUIDs generation, getting current DateTime is a sign of not following idempotency (unique values per every invocation)
* **State Information with Data**: Associate any required state information directly with your data. For example, if processing an order, include the order’s state within the order data itself.

**Practices to Avoid**

* **Avoid Static Variables**: Do not use static variables to store state information. Static variables retain their values between function executions, which can lead to inconsistent behavior and make your function non-scalable.
* **No In-Memory Caching**: Avoid using in-memory caching for stateful data. While caching can improve performance, it can also lead to state inconsistencies if the function scales out to multiple instances.
* **Avoid Local File System**: Do not rely on the local file system to store state or temporary data. The local file system is not shared across instances and can be wiped out if the function app is scaled or restarted.

1. **Avoid Long-Running Functions**: Long-running functions can lead to timeouts and increased costs. Instead, break down complex workflows into smaller, manageable functions. Use Azure Durable Functions to orchestrate complex workflows and manage state across multiple function executions.
2. **Use Retry Patterns**: Implement retry logic to handle transient failures, which are common in cloud environments. Many Azure triggers and bindings already support retry mechanisms.
3. **Defensive Programming**: Write your functions assuming your function could encounter an exception at any time. Design your functions with the ability to continue from a previous fail point during the next execution. This includes validating inputs, handling exceptions, and logging errors for easier troubleshooting.
4. **Optimize Performance**: Monitor and optimize the performance of your functions. Use Azure Application Insights to monitor, find bottlenecks and find opportunities for improvement.

Key Vault

Optional if we have time: talk about how instead of storing potentially sensitive information like connection string in env variables, read it from Key Vault, and permissions required.

1. Instead of reading from Env variable, let's put value into the Key Vault.  
   Create a new key Vault, try to create new secret, but it would fail thanks to RBAC.  
   Add role of "Key Vault Administrator" to yourself - if you are not the admin of your resources  
   Try again: data-connectionstring
2. Add code to read connection string from KeyVault (see repo)

1. Install necessary packages.  
   Remember to add them to the requirements.txt
2. Debug, see that we retrieved a secret. Explain this is because we are logged in and we are admin in for the Key vault, that's why we were given the access to get the credentials.
3. Deploy, try to run again remotely - add see that it does not run. (don't actually wait)
4. Look in the logs - traces (remind that's debugging of deployed functions can be annoying)  
   Show error retrieving the token - because the app itself does not have access to the KeyVault, before we used our right to access it, now it's the app itself trying to make a call and failing. Need to give our function app to access and read key vault.  
   First, we need to give Function App an identity. Then, we need to go to the key vault and give that identity *Key Vault Secrets User* access.

1. Try to run again and see file updated