**Azure Durable Entity Step by Step with Python (hands-on included)**



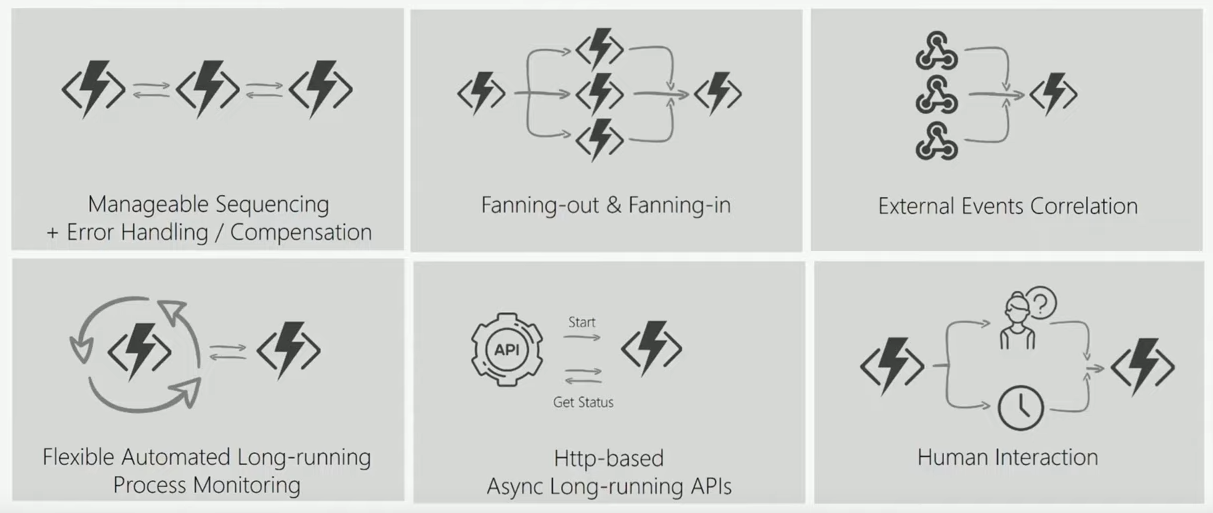
Hola folks! Today I am coming to you with another interesting step-by-step article on Azure Durable Entities. In this article we will be focusing on a special feature released with Azure’s Durable Function 2.0, known as Durable Entities. However, to build up the discussion, I will be touching upon the topics like Azure Functions, Azure Durable Functions and Durable Function elements, before diving into the details and hands-on of Durable Entities.

**What is an Azure Function?**

Azure functions provide serverless solutions for business use cases. They give the developer the freedom to concentrate more on the business logic than the aspects like deployment, infrastructure and maintenance. You can also optimize on cost by paying only for what you use.[1] Azure functions supports the event driven architecture as well. This makes it a good candidate for creating integrated pipelines which combines complex components like Azure Stream Analytics Jobs, Azure EventHubs, Azure ML etc.

**What is a Durable Functions and its ecosystem?**

Azure Durable Functions are an extension of Azure Functions, which gives the capability to create stateful elements in a serverless architecture based solutions. This gives the capability for Azure Functions to manage some of the complex application patterns which otherwise will not be possible. [2]



Application patterns that is facilitated by Durable Functions (Source [3])

The Durable Function supports 4 main Function types with the release of Durable Function 2.0. Language support includes C#, Javascript, Python and PowerShell.



Durable Function types (Source [4])

**Client**

In simple terms they are the ones that enqueue a message to the task hub, acting upon input triggers like HTTP trigger binding, EventHub trigger binding, etc. These enqueued messages then calls upon orchestrator or entity to carry out different tasks. Specialty of client function is its use of the durable client output binding, which may again be HTTP trigger bindings, EventHub trigger bindings, etc.

Apart from these, Client Functions are capable of interacting with running orchestrator or entity. This gives them the capability to query, terminate, etc those orchestrators or entities.

**Orchestrator**

Orchestrators are the ones that orchestrate the workflow. It can carry out different types of activities calling upon various types of elements like activity functions, sub-orchestrations, external events, HTTP, and timers. They can also interact with entity functions. They are stateful and make use of the history table to keep track of their execution.

**Activity**

These are the functions that carry out actual tasks in a workflow, hence, called the “basic unit of work” in Durable function scenario. It can be used to carry out various type of tasks ranging from network calls to CPU heavy tasks. However, important point to notice is that Activity Functions are stateless.

You can refer to Azure Documentation to read more on these. [5]

The 4th type, Entity Functions (also known as Durable Entities or Stateless Entities) is the type that steals the spot light of this article.

**What is a Durable Entity?**

This function type of Azure Durable function is again a stateful one like orchestrators. However, in contrast to orchestrator who manages it state implicitly using control flow, Durable entities manage its state explicitly.

When it comes to the behavior, you can think of Entity and Entity instance as a class and an object in Java. A class provides a blueprint on what its instance can do and what are its properties. Similarly, Entity provides a sort of blueprint, describing what are the operations its instances are capable of. Examples of such operations includes add, get, reset, etc. which is capable of creating, reading, updating, and deleting the state of the entity. [6]

For example, think like you have an Entity called step\_counter. For each user in your application, you need to keep their step count. So this step counter Entity will have many Entity instances which can be identified distinctly using their entity\_key. So for user John Doe, you can have an Entity instance with the key john\_doe.

**Where the state is stored?**

By default, the state of the entity instances are stored in the Azure Storage Account which is attached to the Azure Function we created to host the Durable Entity Function. You don’t have to add any configuration to get this setup. These entity instances does not have any expiration time and there is no numeric limit on how many entity instances you can make. However, there can be limits imposed by the limitations in the storage account side. To prevent conflicts, all operations on a single entity are guaranteed to execute serially.

Currently as a preview feature, there are couple of more storage provider options given to users. Those are Netherite and MSSQL (on-premise servers or Azure SQL). These gives the user option to leverage on things like better throughput or disconnected environments. However, those need to be manually configured. [7]

In my opinion, this is certainly an area which Azure can improve on more. I am especially curious on whether they provide the facility for custom readers to consume from such storage providers directly, without going through the function app.

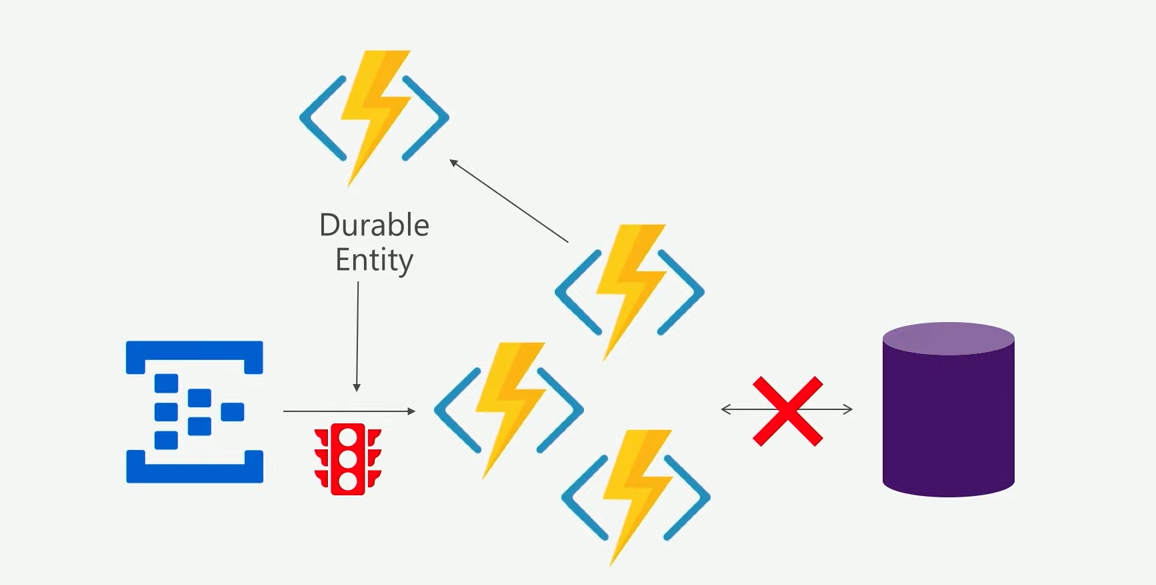
**Access Entities**

* **Calling** an entity uses two-way (round-trip) communication. You send an operation message to the entity, and then wait for the response message before you continue. [6]
* **Signaling** an entity uses one-way (fire and forget) communication. You send an operation message but don’t wait for a response. While the message is guaranteed to be delivered eventually, the sender doesn’t know when and can’t observe any result value or errors. [6]

Who can do what?

* Client: Signal Entity and Read state
* Orchestrator: Signal Entity and/or Call Entity
* Entity: Signal Entity

**Circuit breaker pattern: A special patters that an Entity function can facilitate**



Circuit breaker pattern with Durable Entities (Source [3])

We can create a circuit breaker pattern using a durable entity, to monitor the health of a function app that communicates with a Azure SQL like database, It can take action when there is abnormal number of issues coming from the function app.

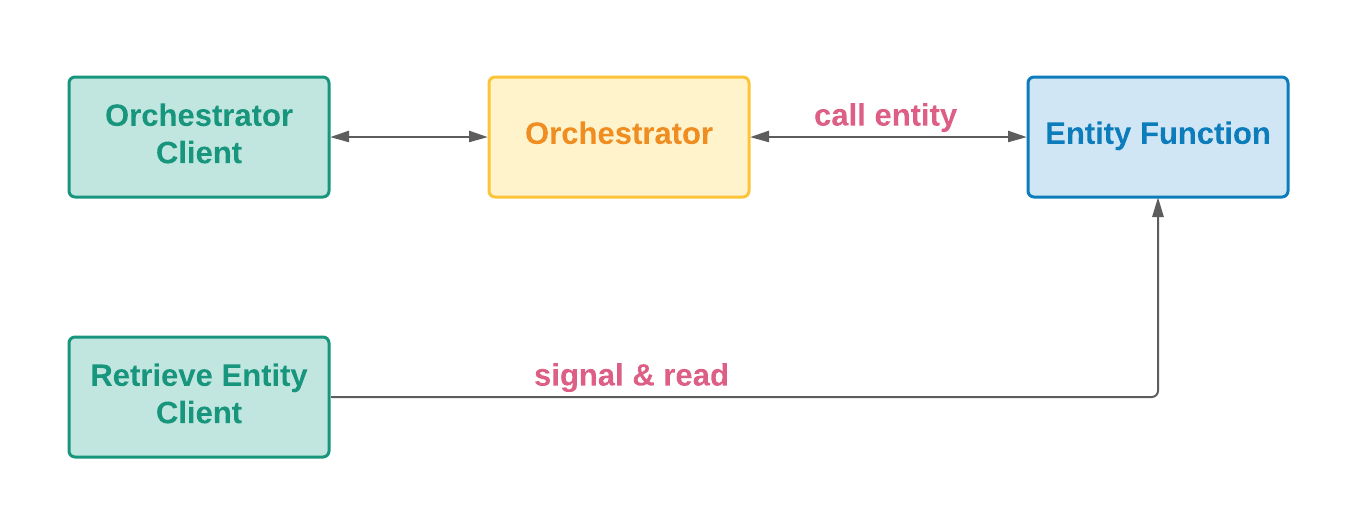
Ex: even though the function app scales, there might be difficulties in scaling for Azure SQL. In such case we can use a circuit breaker durable entity to switch off function app or alert component owners.

**Hands-on**

Lets implement Mortal Kombat style fighting league with Marvel Characters!

**Use case**

Assume we are keeping track of our Marvel Leader Board. It keeps track of not only the total\_combos but also the total\_wins each character had after each round. Then this leader board should have the capability to be reset after a competition. So in this case the state is not a simple int value but rather a dictionary containing those two data fields.



Architecture of the solution for the use case

**Prerequisites**

* An Azure account with an active subscription. (You can [create an account for free](https://azure.microsoft.com/free/?ref=microsoft.com&utm_source=microsoft.com&utm_medium=docs&utm_campaign=visualstudio) as well)
* The [Azure Functions Core Tools](https://docs.microsoft.com/en-us/azure/azure-functions/functions-run-local#install-the-azure-functions-core-tools) version 3.x.
* [Visual Studio Code](https://code.visualstudio.com/) on one of the supported platforms. (I will be using Ubuntu 20.04)
* The [Azure Functions extension](https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-azurefunctions) for Visual Studio Code.

**Create Orchestrator client**

I am not much of a fan of python venv, so lets go ahead with a conda environment. (If you wish to use venv, you can create one directly using Azure Functions extension in vs code)

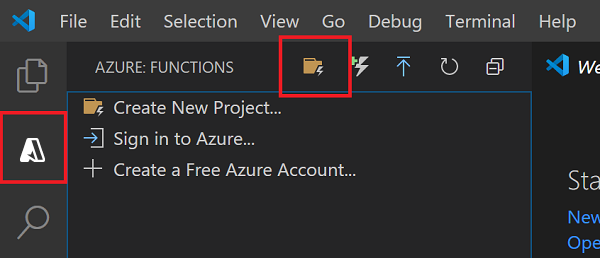
To create Python 3.9 conda environment named azure-durable-entity, with pip installed, use the below command.

conda create -n azure-durable-entity python=3.9 pip

Next activate the environment and install azure-functions and azure-functions-durable.

conda activate azure-durable-entity  
pip install azure-functions azure-functions-durable

Next lets create a new project using the Azure Functions extension which you can find in the Activity bar bearing the Azure icon. Then select the *Create new project* icon.



Create New Project Icon of Azure Function Extension

The interactive interface will prompt you to do few things:

1. Select your project location
2. Select language — in our case, it is Python
3. Select Environment — choose *skip env* option as we already made our environment.
4. Select the template — choose *Durable Functions Http Starter*option
5. Select Authentication level — lets start with *Anonymous* auth level

Vola! Your first step is done. Lets go ahead and have a look at what are the files that got created, and their role in Azure Functions.

Here there will be bunch of files and folders created for you by the Azure Function Extension. Amidst them, there two very important ones like function.json and \_\_init\_\_.py.

The function.json file is a file that is there for each Azure Function (of any of the above mentioned 4 function types) that is in the current project. (Keep in mind that each project may have one or several functions.) It is responsible for defining input bindings and/or output bindings.

The\_\_init\_\_.py is responsible for the function’s core logic. For example, if it is an orchestrator it will describe what activity function or what entity function to call or what to return to the client. If it is an entity function it will describe the operations that it is capable of.

There are many more other files and directories which are not used in this use case. You can read on them more in the Azure Documentations. [8]

Moving back to our use case, make sure the function.json is identical to the one below:

function.json of orchestrator client

In your \_\_init\_\_.py, write the logic of calling the orchestrator as below (I have described the code with comments):

\_\_init\_\_.py of orchestrator client

**Create Orchestrator**

Go to the Azure Function extension and select the ⚡ icon which says “create function”.



Create Function Button

In the drop down you are presented with, select the option, “Durable Functions orchestrator” and give a name. It will add a new folder to your project with the name you provided, which is corresponding to another Azure Function. Its function.json should look similar to the one below.

function.json of orchestrator

Lets write the logic in the \_\_init\_\_ .py (comments explain the code):

\_\_init\_\_.py of orchestrator

**Create Avenger Scoreboard**

Similarly we can then create the Entity Function by clicking the “Create Function” and the selecting the “Durable Function Activity”. In here you will have to update the function.json to respond to entityTrigger.

function.json of Entity Function

Similar to previous functions, lets write our code in the \_\_init\_\_.py.

\_\_init\_\_.py of Entity function

**Create Retrieve Entity HTTP Client**

Lets create another HTTP trigger bind client to retrieve the state of an entity instance. Click on the “Create Function” and the select the “Durable Functions HTTP starter”. Give a name to proceed. The function.json and \_\_init\_\_.py should contain following code. \_\_init\_\_.py contains the concepts discussed above and self-explaining code.

function.json of retrieve-event client

\_\_init\_\_.py of retrieve-event client

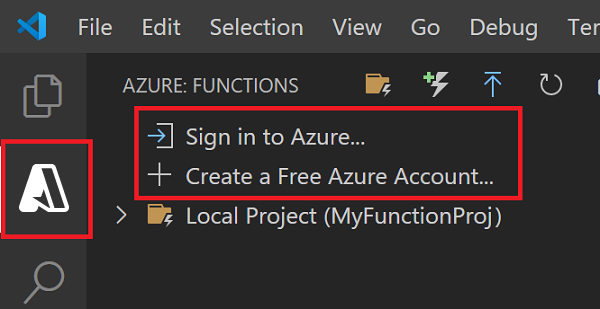
Now we are ready with all our functions. Next step is to create an Azure Function and publish the project.

**Create Azure Function to host your project**

This part is pretty straight forward and described in Azure documentation: Create your first function in the Azure portal (Only do the steps mentioned in the “[Create a Function App](https://docs.microsoft.com/en-us/azure/azure-functions/functions-create-function-app-portal#create-a-function-app)” section). So I won’t go into the details of it. But if you have any questions, leave a comment down below, I will help.

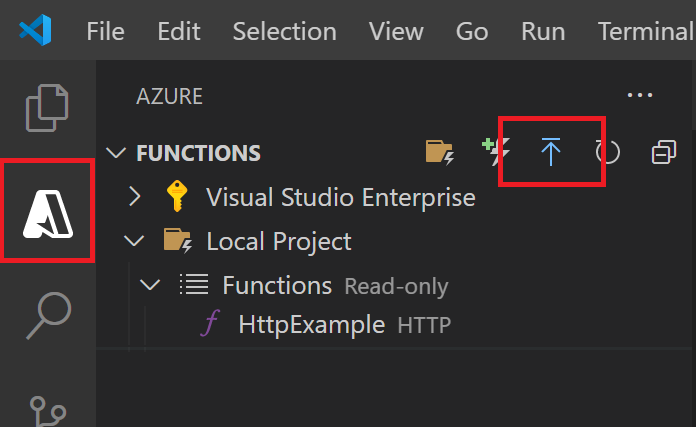
**Publish to Azure**

In the Activity bar go to Azure Functions extension. Then in the Azure Functions area, first select the *Sign in to Azure* option.

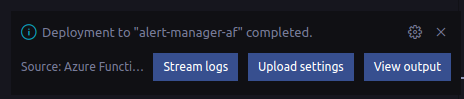


You will be navigated to a browser, where you can enter your credentials and login to Azure.

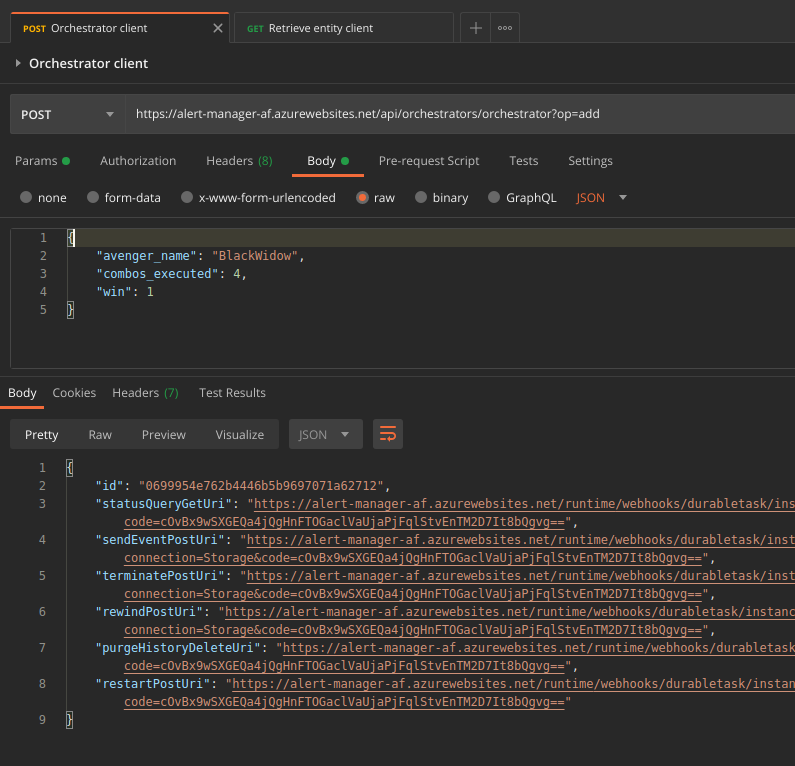
Publish the changes to the Azure Function you created previously.



Wait for a notification similar to the one below:

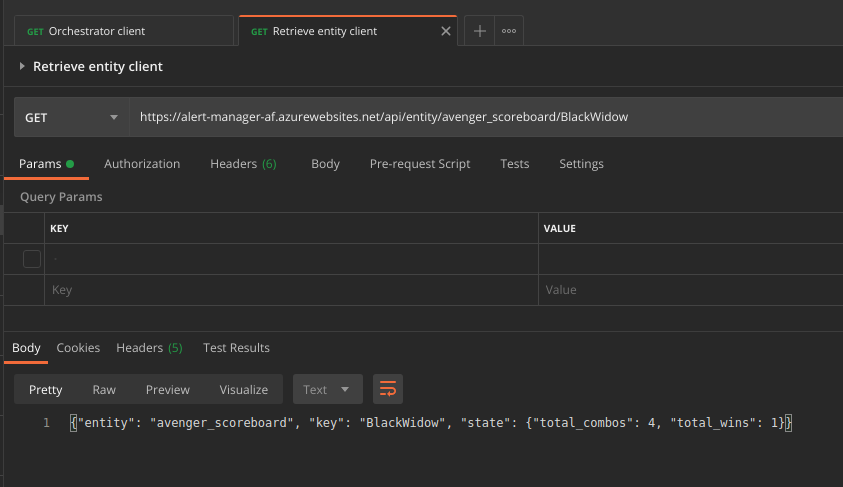


Lets send a request in postman and see:



You can see that I am mentioning the operation as “add” in the request parameters while adding the data as a JSON in the request body. Here we are getting status code 202 indicating that our entity instance is created. Also if you click on the statusQueryGetUri, it will give you the status of the orchestration you initiated.

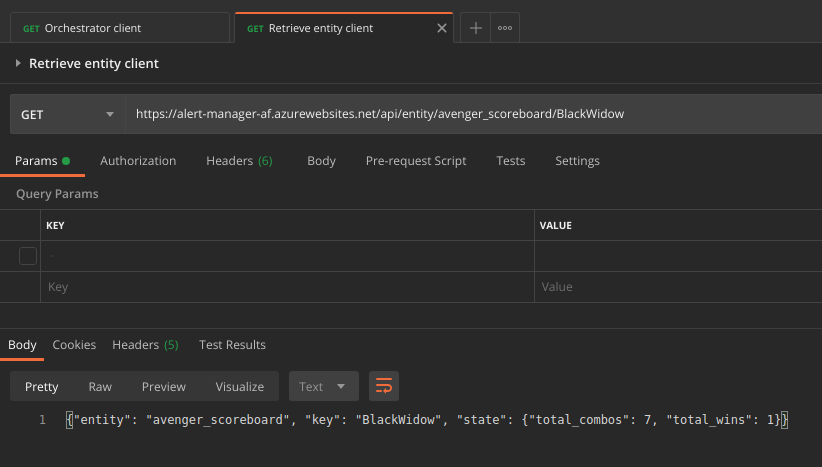
Now lets try retrieving the entity instance we created:



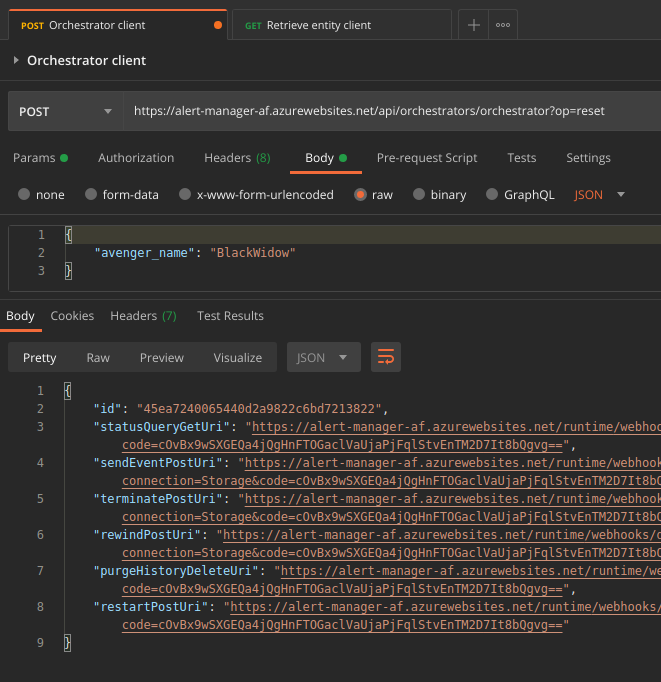
Send another

{"avenger\_name": "BlackWidow","combos\_executed": 3,"win": 0}

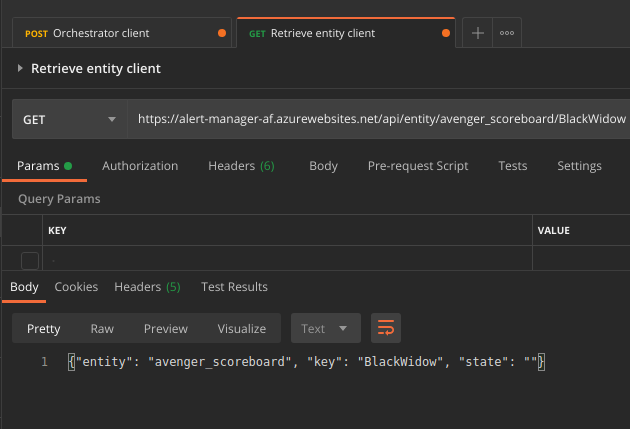
It will return the following



You can try the reset operation as well:



Now if you retrieve the same object, you can see that the state is empty.



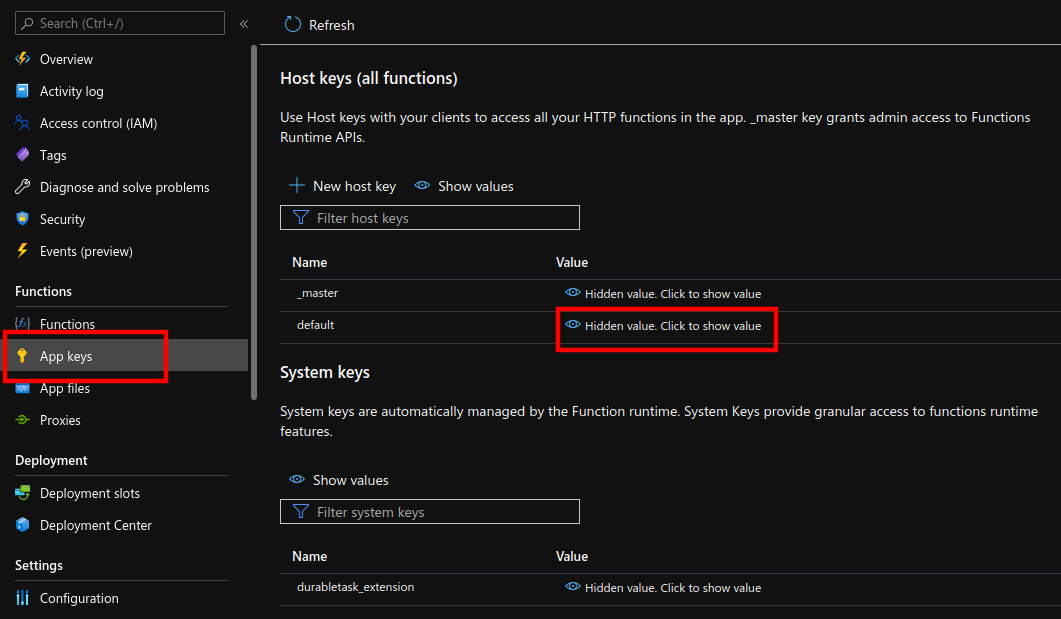
**Bonus: Authentication**

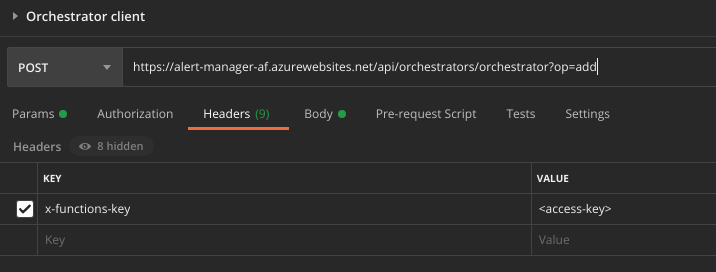
There are 3 types of Authentication available in Azure Functions and Azure Durable Functions.

* Function: Provide function level access using keys. This can be either for each function or to all functions as a whole.
* Admin: Provide administrative access to the runtime REST APIs apart from providing function level access. This is also by using access keys.
* Anonymous: Does not have any access control

Under the folder of HTTP client function which you want to add authentication, goto function.json file. Under the bindings section, change the authLevel from anonymous to function.

To retrieve the Access Key to the Function, go to your function app in Azure Portal. In the left plane, under the Functions section, select the Access Keys category. It will bring up some access keys for the function app. You can unhide and copy the access key named default, and use it as a header in the request to access the auth enabled function. The header key should be x-functions-key.





So that’s it for the today’s article. I hope you enjoyed it. I started it more as a self note, even though I decided to publish later. So if there is something unclear, please comment down below, I will surely reply. You can find the full project in the following Github repository:

<https://github.com/savindi-wijenayaka/azure-durable-entity>