# [Proctor Guide]

# Challenge 05 – Weather Data

## Task 1 – Create a Dark Sky account

This is very straightforward. Attendees just need to create a free account and then go to their account page (<https://darksky.net/dev/account>) to access their API key for later use.

## Task 2 – Prepare weather API request input data

Attendees should write a list of records out of the single Hive table they created to complete Challenge 02 – i.e. the single table that merges all Yellow and Green data. If they did not finish Challenge 02, that’s fine; they can use any data subset.

The minimum data they need to export is three fields: date/time, longitude, latitude. They should filter out null, empty, or zero values for both longitude and latitude.

We suggest they use the pickup versions of these fields. Why? Because for eventual (not in this hack) ML work, the weather conditions at pickup are likely more correlated with other trip features and outcomes.

There is a sample Jupyter notebook in Assets/Challenge05/15-ExportWeatherDataInputs.scala to do this. Attendees should conclude this task with a CSV file in a storage account/container which they will access in later tasks.

## Task 3 – Prepare Cosmos DB repository

Attendees should prepare a Cosmos DB database and collection. No partition key is necessary and the minimum specs of 10GB and 400RU are plenty, since we are limited to 1,000 records with DarkSky’s free limit.

## Task 4 – Create an Azure Function App to query the DarkSky API and store weather data

Attendees have to create a Function App that does the following:

1. Reads in the CSV file from Azure storage – this is the file they created in task 2
   1. Of course they could write, and read, the file in other formats than CSV. The reference implementation available in Assets/Challenge05 was written for CSV.
2. Reads each line and parses it into date/time, longitude, and latitude
3. This piece will likely happen after they initially fail when calling the DarkSky API – encourage them to read the DarkSky API doc ***very closely*** and to think about data formats! They may need to change the format of the date/time emitted from Databricks/Hive in task 2 to eliminate fractional seconds, which DarkSky does not accept even though ISO8601 supports them – or they may choose to serialize the date/time to ticks (i.e. seconds since 1/1/1970 – UNIX standard format) though this will take a little more work.
4. Get the HTTP response from DarkSky and extract the content.
   1. By default, the content is very verbose. Suggest to attendees that they review the DarkSky API docs for ways to reduce the response content size. They can add an “exclude” query string parameter with various DarkSky content parts to exclude from the response. I used this:  
      ?exclude=currently,minutely,hourly,alerts,flags
5. Another tricky piece of thinking ahead. In the Attendee doc, Task 4 hints “In addition to weather data, don’t forget to think about what you need to store in Cosmos DB to enable you to later combine the right weather data with each taxi trip”.  
   This refers to the fact that DarkSky’s API response will include longitude and latitude… and the date/time as passed to it, which will not exactly match what is in Databricks! (See point 3.)  
   Attendees should therefore consider augmenting the weather JSON data with another key/value pair, such as “pickup\_datetime”:”[the original date/time before they remove fractional seconds]”.   
   They may even consider writing all three original key-value pairs into the JSON from the weather API, just to make absolutely sure they have identical values as the source data on which to join.  
   The point here is, if they run into difficulty merging the weather data with the taxi trip data by joining on date/time, longitude, and latitude, they should closely examine these fields’ data on both sides and consider adding code in the Azure Function to make this easier.
6. Write the weather data to Cosmos DB.  
   Note that attendees may decide they want to delete all data in Cosmos DB so they can “start clean” with the API. The fastest way to do this is to just delete the Cosmos DB collection and create a new one with the same name. If they do this, and they are using Azure Storage Explorer, they will need to refresh the entire Cosmos DB database node in order to see data in the same-named new collection.

Writing to Cosmos DB is not as straightforward as it sounds!! Here’s why.

Say you create an Azure Function, maybe with a blob trigger (useful in this Challenge). That input will contain more than one record, and each record yields an API response with weather data.

Each of those weather data responses must be written to Cosmos DB. So we can just add a Cosmos DB output to the function and write to the out variable, right? Well… yes, but only once per function execution! So even if we get 100 input records and 100 weather API responses, we can only write to the output variable once… and we can’t even do that within a loop or a conditional statement.

So what’s the answer? Multiple functions, loosely coupled to each other with a queue.

The first function reads in the CSV file and, for each row in it, gets weather data from the API and augments it – as described above. This function then writes each of these weather data documents to a storage queue… and it’s done.

The second function has a queue trigger. When an item hits the queue it’s listening to, the function gets triggered – once for each item on the queue. Our items from the first function are each individual weather data response documents.

Therefore, since the input to this function is a single data item, we can add a Cosmos DB output to this function and write to the out variable unambiguously.

In this way, we reduce the functional responsibility of each function, and we loosely couple the functions with a queue so that neither major task (get data from an API, write data to a database) is constrained by the other.

There is a reference implementation of an Azure Function App with two Functions in Assets/Challenge05/AzureFunctionApp/. Both Functions are written in C#.

IMPORTANT: Azure Function Apps, by default, are instantiated on Functions runtime v1. Some attendees may be tempted to change to runtime v2. If they do this, they will no longer be able to add Cosmos DB outputs (not supported on v2 yet). If attendees ask how they are supposed to write to Cosmos DB when no such outputs are available, ask if they have switched to Functions runtime v2 and, if so, smile and encourage them to try reverting to default Function settings.

The first Function, GetWeather, handles points 1-5 above. The second Function, SaveToCosmosDB, is a simple pass-through that dequeues individual weather records (which are already augmented JSON from GetWeather) and writes them to Cosmos DB.

## Task 5 – connect to the weather data in Cosmos DB from Power BI and create an enriched visualization

Power BI has preview connectivity to Cosmos DB.

Attendees will need to work with the JSON structure in Cosmos DB to join it appropriately to taxi trip data by pickup\_datetime, pickup\_longitude, and pickup\_latitude so that taxi trips can show weather conditions at the date/time and location of pickup.

Attendees will need to navigate the weather data JSON hierarchy. The smaller size data they store (see point above about DarkSky API exclude URL query string parameter) the easier this will be in Power BI.

## Notes

1. Attendees will need to create a Cosmos DB database and collection
   1. No partition key, 10GB, 400RU (minimum specs) are fine
2. They will need to create a storage queue if they implement multiple functions and decouple them, as I did
3. Strongly recommended: use the same storage account they were using for Databricks, and where they wrote the weather input data in Task 2 of this Challenge.
4. They will need to create an Azure Function App. Consumption plan is fine.
5. The first Function, GetWeather, is NOT what I would call production quality code… It’s hack-level (get this done as fast as possible) but at least it’s commented 😊.
6. Throughout both Functions, I left a token [PROVIDE] where – if you give attendees the code – they will need to substitute their own information (e.g. storage account name/key, etc.)
7. Azure Storage Explorer works with both Azure Storage as well as Cosmos DB. If attendees ask how they can manage Cosmos DB data, this tool is the best quick answer.