▪

# Coursework/1

## Contents

## Coursework 1 - Part 1 Specification

- Coursework 1 Part 1: Azure and Python Functions
- Deadline: Wednesday 8th November 12:00 (Week 6)
- Expected effort: 20 hours per student
- Weighting: 20% of module evaluation

In Quiplash, players have profiles with their usernames, games played and total score across all games they have played. They can create their own lists of “prompts” that are used when they play with others.

A game instance is a list of prompts that are shown sequentially to players. Players score in the round is the number of votes their answer received. Player’s total score is the sum of their scores across all rounds. The player with the highest score after all prompts have been answered and voted is the winner.

### Learning Objectives:

- L1: Write Azure Functions in Python (v1 programming model)
- L2: Deploy a FunctionApp on Azure.
- L3: Perform CRUD of documents in Cosmos DB
- L4: Query collections in Cosmos DB

- L5: Configure and use additional Cloud services in the context of a FunctionApp

Marking Scheme:

Marking is by correctness of automated tests. Each question specification includes mark values per passed test. A summary table is below:

| Component | LOs | Marks |
|---|---|---|
| /player/register<br>/player/update<br>/prompt/delete | L1, L3 | 6/20 |
| /prompt/create | L1, L3, L5 | 4/20 |
| /player/login<br>/utils/leaderboard<br>/utils/get | L1, L4 | 6/20 |
| Deployment | L2 | 4/20 |

Handin link:

TBD

Goal

The goal of this part is is to implement the backend of a the quiplash application (player profiles and prompts) using Azure functions (v1) coded in Python and CosmosDB as persistence layer.

You will submit your development version to hand-in and deploy a CosmosDB service and your app in your Azure account.

Schema and Container specifications:

Player:

- Username (unique), minimum 4 characters, maximum, 14 characters
- Password, minimum 10 characters, maximum, 20 characters
- games_played: Counter of games played by this player. Integer >=0.
- total_score: accumulator of score of all games played by this player. Integer >=0.

Example JSON document representing a Player:

```
{"id": "auto-gen-by-Cosmos", "username" : "py_luis", "password": "pythonrulz" , "games_played" :
542 , "total_score" : 3744   }
```

Prompt:

- Username: player that created this Prompt
- Texts: Unordered list of pairs {language, text} of a Prompt and its translations to multiple languages.
  - Language is a code on the table of languages supported by Azure Translation (https://learn.microsoft.com/en-us/azure/ai-services/translator/language-support). In this coursework we focus on ["en", "es" ]
  - Text is a string of minimum 15 characters, maximum 80 characters

Example JSON document representing a Prompt:

```
{"id": "auto-gen-by-Cosmos" ,"username": "py_luis" , "texts" : [{"language" : "en", "text": "The
most useless Python one-line program" }]}
```

You must create a single CosmosDB database named "quiplash" in your Azure account and set up two containers named "player" and "prompt".

The player container must use "id" as partition key.

The prompt container must use "username" as partition key

## Functions to implement

All functions receive as input a JSON document and return a JSON document.

For all Functions, assume the input is well-formed, that is, no need to validate the format of the JSON input. Some functions do ask to perform some validation in the content of the input (e.g. length of a password).

### /player/register/

POST method

(2/20 marks)

```
Input:

{"username":  "string" , "password" : "string"}
```

```
Output: one of the following:

{"result" : true, "msg": "OK" } if player successfully registered. games_played and total_score
must be set to 0 in the DB
{"result": false, "msg": "Username already exists" } if username already exists
{"result": false, "msg": "Username less than 4 characters or more than 14 characters"  } if
username below/above the limit
{"result": false, "msg": "Password less than 10 characters or more than 20 characters"  } if
password below/above the limit
```

Expected tests:

- For correct registering,
  - The right formatted item must be in the DB.
  - Must be able to insert when DB is either empty or not empty
- For validation:
  - Interval and boundary test.

- Messages comply with specification (watch out for typos and blank spaces!)
- Assume we only test one validation at a a time, e.g., no test case will include multiple invalid input

Marking:

- 1.25 marks if player with valid username and password can be successfully registered and player documents stored in the correct format.
- 0.25 marks for each of the three validations
- -0.5 if a message has a typo or you return the wrong boolean result (not cumulable).

## /player/login/

GET method

(1/20 marks)

```
Input:
{"username": "username" , "password" : "pwd"}
```

```
Output:

{"result": true , "msg" : "OK"} if player with input username exists and password matches the
one stored in the DB for that username.

{"result": false , "msg": "Username or password incorrect"} otherwise
```

Expected tests:

- Login of existing user with right password
- Combinations of inexistent/existent usernames with existent/wrong passwords

Marking:

- 0.5 marks for successful login with valid username/password
- 0.5 marks for successful validation of incorrect username/passwords
- -0.25 if a message has a typo or you return the wrong boolean result (not cumulable).

## /player/update/

PUT method

(1/20 marks)

```
Input:

{"username": "user_to_modify" , "password": "pwd" , "add_to_games_played": int ,
"add_to_score" : int }  adds "add_to_games_played" to player's "games_played" and "add_to_score"
to player's total_score.
```

```
Output:

{"result": false, "msg": "Player does not exist" } if username does not exist

{"result" : true, "msg": "OK" } if update executed successfully
```

Expected tests:

- Interval and boundary tests
- Check you really updated what you should.

Marking:

- 0.75 marks for successful update
- 0.25 marks for successful validation of player existence
- -0.25 if a message has a typo or you return the wrong boolean result (not cumulable).

/prompt/create

POST method

(4/20 marks)

Configure the Azure Text Translation service (https://learn.microsoft.com/en-us/azure/ai-services/translator/text-translation-overview) in your account (don't forget to set the free tier and be mindful of its quota!) and use it to implement the following

```
Input:
{"text": "string", "username": "string" }
```

```
Output:

{"result" : true, "msg": "OK" } on successful prompt creation

{"result" : false, "msg": "Player does not exist"} if player with input username is not in the
DB

{"result": false, "msg": "Prompt less than 15 characters or more than 80 characters"  } if text
below/above the limit

{"result": false, "msg": "Unsupported language"} if the language detected by Azure Translation
Service is not supported for translation, or the confidence of the detection is less than 0.3
```

Expected testing:

- A prompt in each language. The languages the quiplash app supports are English, Spanish, Italian, Swedish, Russian, Indonesian, Bulgarian and Chinese Simplified
- A prompt in an unsupported language.
- A prompt with garbled words to simulate low confidence of detection
- Assume we only test one validation at a a time, e.g., no test case will include multiple invalid input

Marking:

- 2 mark for correct translation to all languages the quiplash app supports
- 1.5 mark for correct detection of unsupported language
- 0.25 mark for validation of player existence
- 0.25 mark for validation of prompt's length
- -0.5 if a message has a typo or you return the wrong boolean result (not cumulable).

/prompt/delete/

POST method

(3/20 marks)

```
Input:

{"player" : "username" } deletes all prompts authored by player "username"
or
{"word" : "offensive-word" } deletes all prompts that include "offensive-word" (case-sensitive)
in its english language text. Do not delete if "word" is substring of a word in the prompt,
Example: if "word" = "boomer", the prompt "Where are all the ka-boomers!" must not be deleted
```

```
For Collection state ( en/es texts only for conciseness):

[{"id": "auto-gen-1" , "username": "py_luis",
  "texts": [{"text" : "The most useless Python one-line program", "language" : "en"} ,
                                   {"text" : "El programa de una línea en Python más inútil",
"language" : "es"} ]
 },
 {"id": "auto-gen-2" ,"username": "py_luis",
  "texts": [{"text" : "Why the millenial crossed the avenue?", "language" : "en"},
                                   {"text" : "¿Por qué el millenial cruzó la avenida?",
"language" : "es"} ]
 },
 {"id": "auto-gen-3" ,"username": "js_packer",
   "texts": [{"text" : "Why the ka-boomer crossed the road?", "language" : "en"},
                                   {"text" : "¿Por qué el ka-boomer cruzó la calle?", "language"
: "es"} ]
 },
 {"id": "auto-gen-4" ,"username": "les_cobol",
   "texts": [{"text" : "Why the boomer crossed the road?", "language" : "en"},
                                   {"text" : "¿Por qué el boomer cruzó la calle?", "language" :
"es"} ]
 }
]

Input: {"player" : "py_luis"}

Output: {"result": true, "msg": "X prompts deleted"} with X the number of prompts deleted


New Collection state:

[ {"id": "auto-gen-3" ,"username": "js_packer",
   "texts": [{"text" : "Why the ka-boomer crossed the road?", "language" : "en"},
                                   {"text" : "¿Por qué el ka-boomer cruzó la calle?", "language"
: "es"} ]
 },
 {"id": "auto-gen-4" ,"username": "les_cobol",
   "texts": [{"text" : "Why the boomer crossed the road?", "language" : "en"},
                                   {"text" : "¿Por qué el boomer cruzó la calle?", "language" :
"es"} ]
 }
]

Input: {"word" : "boomer"}

Output: {"result": true, "msg": "X prompts deleted"} with X the number of prompts deleted

New Collection state:

[ {"id": "auto-gen-1" , "username": "py_luis",
  "texts": [{"text" : "The most useless Python one-line program", "language" : "en"} ,
                                   {"text" : "El programa de una línea en Python más inútil",
"language" : "es"} ]
 },
 {"id": "auto-gen-2" ,"username": "py_luis",
  "texts": [{"text" : "Why the millenial crossed the avenue?", "language" : "en"},
                                  {"text" : "¿Por qué el millenial cruzó la avenida?",
"language" : "es"} ]
 },
 {"id": "auto-gen-3" ,"username": "js_packer",
   "texts": [{"text" : "Why the ka-boomer crossed the road?", "language" : "en"},
                                  {"text" : "¿Por qué el ka-boomer cruzó la calle?", "language"
: "es"} ]
 }
]
```

Expected testing:

- For "player" input:
  - Like example above, a state and input where a deletion has to be done
  - Assume "player" exists, no need to check existence like in prompt/create
- For "word" input
  - A substring case like above "boomer"/"ka-boomer" to check you delete what you should
  - A word that does not exist in any prompt (expected output: no change in the collection)
  - A word that exist in at least one text in a language different than english , but does not exist in any english text (expected output: no change in the collection)
- Assume test input is an existing username (no need to check it exists)

Marking:

- 1 mark for "player" input
- 2 marks for "word" input
- -0.5 if a message has a typo or you return the wrong boolean result (not cumulable).
-

/utils/get

GET method

(2/20 marks)

```
Input:

{"players":  [list of usernames], "language": "langcode"} return a list of all prompts' texts in
"langcode" language created by the players in the "players" list.
If a player in "players" does not exist or does not have any prompt , do not return error,
return prompts from users that do exist and have prompts.
If none of the usernames in the list exist or have prompts, return an empty list.
Output can be in any order.
```

```
For Collection state ( en/es texts only for conciseness):

[{"id": "auto-gen-1" , "username": "py_luis",
  "texts": [{"text" : "The most useless Python one-line program", "language" : "en"} ,
                                  {"text" : "El programa de una línea en Python más inútil",
"language" : "es"} ]
 },
 {"id": "auto-gen-2" ,"username": "py_luis",
  "texts": [{"text" : "Why the millenial crossed the avenue?", "language" : "en"},
                                  {"text" : "¿Por qué el millenial cruzó la avenida?",
"language" : "es"} ]
 },
  {"id": "auto-gen-3" ,"username": "js_packer",
    "texts": [{"text" : "Why the boomer crossed the road?", "language" : "en"},
                                 {"text" : "¿Por qué el boomer cruzó la calle?", "language" :
"es"} ]
  },
  {"id": "auto-gen-4" ,"username": "les_cobol",
    "texts": [{"text" : "Why the boomer crossed the road?", "language" : "en"},
                                 {"text" : "¿Por qué el boomer cruzó la calle?", "language" :
"es"} ]
  }
]


Input: {"players" : ["js_packer","les_cobol"], "language": "en"}

Output:

[
{"id": "auto-gen-4" , "text" : "Why the boomer crossed the road?" , "username" : "les_cobol"},
{"id": "auto-gen-3" , "text" : "Why the boomer crossed the road?" , "username" : "js_packer"}
 ]
```

Expected Testing:

- Similar to example above

Marking:

- 2 marks for "players" input
- -0.5 if if you return a wrongly formatted result

/utils/leaderboard/

GET method

(3/20 marks)

```
Input:
{"top" : k }  k an integer value > 0 and <= size of prompt collection
```

Outputs list of k players with highest total_score. In case two or more players have the same total_score, apply tie breaks in the following order:

1. Player with less games_played go first.
2. Alphabetic order

```
For Collection state:

[{"username": "A-player", "games_played" : 10, "total_score": 40} ,
{"username": "B-player", "games_played" : 10, "total_score": 40} ,
{"username": "C-player", "games_played" : 20, "total_score": 80} ,
{"username": "D-player", "games_played" : 10, "total_score": 80} ,
{"username": "X-player", "games_played" : 50, "total_score": 100} ,
{"username": "Y-player", "games_played" : 10, "total_score": 40} ,
{"username": "Z-player", "games_played" : 1, "total_score": 10} ,
]

Input: {"top" : 5}

Output:
[ {"username": "X-player", "games_played" : 50, "total_score": 100} ,
  {"username": "D-player", "games_played" : 10, "total_score": 80} ,
  {"username": "C-player", "games_played" : 20, "total_score": 80} ,
  {"username": "A-player", "games_played" : 10, "total_score": 40} ,
  {"username": "B-player", "games_played" : 10, "total_score": 40} ,
]

Note:
 the games_played tiebreak between "D-player" and "C-player"
 the alphabetic tiebreak between "A-Player" and "B-Player" (same score)
 "Y-player" is discarded because the tiebreaks put them below the "top" value
```

Expected testing:

- A state and input that does not need tiebreaks
- A state and input that only needs games_played tiebreak
- A state and input that needs both tiebreaks (e.g. the example above)
- All 3 cases above will be re-tested with a state including a player like "Y-player"
- No need to validate that "top" is in the right range.

Marking:

- 1 mark for correct output when tiebreaks are not needed and "Y-player" is correctly discarded
- 1 mark for correct output when only games_played tiebreak is needed and "Y-player" is correctly discarded
- 1 mark for correct output when both tiebreaks are needed and "Y-Player" is correctly discarded
- -0.5 if a message has a typo or you return the wrong boolean result (not cumulable).

Hint: We encourage the use of Composite Indexes, an advanced technique, but the way you would do this in real life due to efficiency. However, as this was not covered in lectures, other solutions are allowed.

Deployment

(4/20 marks)

You will deploy your application on your Azure account, and update the provided wrapper.py file adding the URIs of your Azure Functions server and the Function App key.

Expected test:

We will run the same test script that we run on your development version on your deployment to check that we can access all functions and they give the same results as your submitted version, in other words, that you managed to deploy your app.

Marking:

- 4 marks if all implemented functions are accessible on your deployment and functions you didn't implement return the output {"msg" : "not implemented"}. Note this means that you have to have the correct function names, the correct routes, and the correct authorisation. There is a -0.5 penalty for each function that can't be run on deployment. Reasons for failure here include but are not limited to:
  - A typo in the expected name of the function or in the route.
  - Wrong type of HTTP method

        - When not implemented in local development submission, throws an error or exception instead of {"msg" : "not implemented"}
        - Runs on the local development submission but the deployment throws an error, exception or returns a different result.
        - A function that is incorrect or partially correct is not penalised in this criterion, provided it returns the same (incorrect) values than the one on your local development version .
    - -2 marks penalty if your local.settings.json is missing a necessary credential. We will get in touch with you to provide it, your mark will be delivered later than the others.
    - 0 marks in this criterion if your app has a date of last deployment after the deadline (Instructions to prove this will be provided in due course, it's a couple of parameters you need to provide in your local.settings.json so we can access this log in your Azure account).
    - 0 marks in this criterion if you don't configure the specified partition keys or otherwise mess up your CosmosDB so markers can't access it. This breaks the test script because then we can't inject data in the collections for the tests. Marking is deferred and takes place on marker's Azure account.
        - If you get the composite indexes for /utils/leaderboard wrong, it only affects that question, so you don't get penalised here.
        - If you misconfigure the Translation service for /prompt/create, it only affects that question, so you don't get penalised here.

## What do you need to hand in

A zipped folder named "quiplash" containing the development version of your FunctionApp. A marker with Azure tools installed should be able to unzip it and run 'func start' right away.

The expected structure of the folder follows the Azure recommendation and is given below:

```
quiplash/
 | - .venv/ #Should remove it, but if it's there it's fine, we do.
 | - .vscode/ #Only there if you use vscode, should remove it, but if you forget it's fine, we do.
 | - player_register/ #All functions must include init and function.json (omitted in the remainder for conciseness)
 | | - __init__.py
 | | - function.json
 | - player_login/
 | - player_update/
 | - prompt_create/
 | - prompt_delete/
 | - get/
 | - leaderboard/
 | - shared_code/ #You may define helper functions here
 | | - __init__.py
 | | - your_helper_function.py
 | - tests/ #Your own tests, encouraged, but won't be marked
 | | - test_player_register.py
 | - .funcignore
 | - host.json
 | - local.settings.json
 | - requirements.txt
```

A local.settings.json template will be provided with the credentials you need to provide for marking to work. [TBD after week 3]

## How marking works

1. Marker downloads your local development version and checks local.settings.json has all necessary credentials to run it against your deployed CosmosDB and Translation Service. If failed, defer to secondary assessment
2. Marker starts development server and runs automated test script 1. Script 1 marks and provides as feedback the test case that led to the failure.
3. Marker checks Script output is consistent. Adds any additional comments if required.
4. Marker uses automated script 2 to check against the app deployed in student's Azure account the criterion on section deployment.