Cloud Computing 2022/23 spring

Practice 4 - Cloud Functions

In this lab, we will use Azure Cloud and take a look at the **Function App** service, which can be considered a **Function as a Service** (FaaS) or **Serverless** platform.

In **FaaS** Cloud Computing model your application consists of a set of functions or microservices. **Cloud Functions** are not constantly running in the background. They are executed only when specific user-specified **events** are triggered (e.g. when a new file is uploaded, a new entry is added to the database, a new message arrives, etc.).

You will use the existing account in **Azure Cloud** and set up multiple **Python Cloud Functions** which are automatically triggered on either web requests or database events. You will also use the Azure **Cosmos DB** NoSQL service as the database for your **Cloud Functions**. In addition, You will make a new Twilio trial account and use the Twilio SMS API for sending messages to mobile phones.

Your task will be to create three Serverless functions:

1. **htmlForm** - This Function will display the message board page, enable adding new messages and display current messages in the database
2. **handleMessage** - This will act as a backend function for entering new messages into the Cosmos DB once the user clicks the submit button on the web page
3. **cosmosTwilioTrigger** - This function will be triggered every time there is a new message added to the Cosmos DB. It will send a message to Twilio API to send a new SMS to your own mobile phone number.

Additional materials, tutorials, and references

- Azure Cosmos DB - https://learn.microsoft.com/en-us/azure/cosmos-db/
- Azure Functions Python reference - https://learn.microsoft.com/en-us/azure/azure-functions/functions-reference-python
- Azure Functions Python examples - https://learn.microsoft.com/en-us/samples/browse/?products=azure-functions&languages=python
- Introduction to JSON - https://www.digitalocean.com/community/tutorials/an-introduction-to-json

Exercise 4.1. Creating your first Cloud Function

Let's first create a Functions App, which acts as a group or a logical container for your Cloud Functions. The rest of the functions in this lab will be created under this Function App.

- From the Azure Portal (https://portal.azure.com/), search for "**Function App**" or go to the Cloud Functions console directly
- Create a new Function App
- Choose **Azure for Students** subscription
- Choose function App name: should include your last name
- It should NOT include any special characters.
- Choose Python for the Runtime stack, and `Python 3.9` as the `Version`
- Choose North EU for the region.
- Plan type should stay Consumption (Serverless)
- Either use an existing or create a new Resource group and Storage account (If you deleted them in the last lab)
- As a result, a container for functions will be created for you.

Let's now create our first function

- Under the Functions App, move to the **Functions** page and create new Function.
- Choose: HTTP trigger
- Scroll down and name it: **htmlForm**
- Authorization level MUST be **Anonymous**
    - This allows anyone from the internet to access the address of the function without credentials.
- Once the function is created, check the **Overview page** and use the **Get Function Url** to copy the link to the web function.
    - Open this link in a browser to test out the generated function
    - Add **?name=User** to the function url to have it print out the provided user name.

Let's modify the Function code to generate the HTML document for a message board. Similarly to how the previous lab's Flask-based message board application worked. This function implements the front-end for our application. We can edit and test the function through our browser. We do not need to use source code repositories, but it would be best practice to use them in real-life scenarios for version control and automation of Cloud Functions.

- Open **Code** + **Test** page of the new function
- You can remove most of the code inside the `def main(...):` method. But keep one `func.HttpResponse()` method call as an example and also the `logging.info()` method. Feel free to add additional logging statements for debugging.
- Let's create an HTML page string object in Python at the start of the `def main(...):` method:

```
html_data = """
  <title>Message board</title>
  <body>
    <h4> Welcome to the message board. </h4>
    <h4> Enter a new message</h4>
    <form action="/api/handleMessage">
        <label> Your message: </label><br>
        <input type="text" name="msg"><br>
        <input type="submit" value="Submit">
    </form>
    </body>
    </html>
"""
```

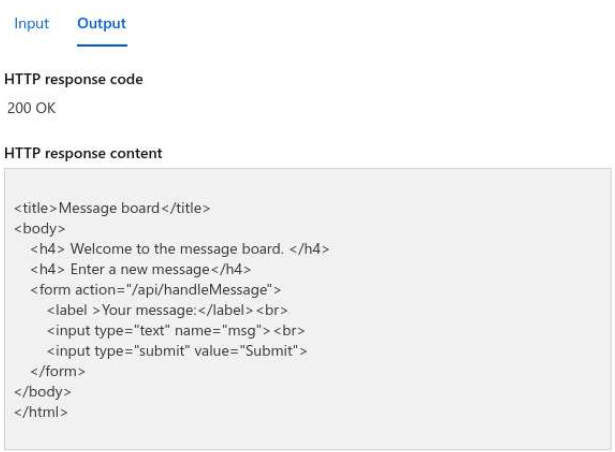
- It contains the HTML form for submitting messages to our message board.
- Modify the function to return the HTML document content using the `func.HttpResponse()` method at the end of the main() method:

```
return func.HttpResponse(
    html_data,
    status_code=200,
    mimetype="text/html"
)
```

- Save the function code.
- You should note that the HTML form links to an address of: `/api/handleMessage`
    - This is an address for another Serverless function we create in the next task.
- Be very careful with Python's code Indentation.
    - Always use the same number of spaces to separate the code lines at the same level.
    - Guide: https://peps.python.org/pep-0008/#indentation)
    - It is easy to make mistakes when editing raw code in the web interface instead of IDE.

Let's now test that the function works

- Click on Test/Run (top row)
- Change the HTTP method to GET
- Click Run.
- In some later tasks, you may need to provide additional parameters, then you can edit the request Body or query parameters.
- The result should look something like this:



- Go back to Function Overview, get the function Url address, and open it in the browser to test that the HTML page works properly.
    - The result should look something like this:

- If you click on the Submit button, it will not work properly because the HTML form links to a different function (`handleMessage`) in the SAME Function app, that we have not yet created yet.
    - The second function will put the message into a Cloud database and later we will update the currently created `htmlForm` function to also list a set of messages from the database.

- Go back and modify the function to also display your name somewhere inside the Web page.

Exercise 4.2. Setting up Cosmos DB cloud database

Before we create the second Serverless function (`handleMessage`), we need to set up a Cloud database. In this task, we will use the Azure Cosmos DB - NoSQL database.

- In the top Azure portal search bar, search for **Azure Cosmos DB** and create a new DB.
- Choose: Azure Cosmos DB for NoSQL
- Choose the exact same Resource group as you used for the Function App.
- Also choose the same Location
- Freely choose the name for the account, but do not use any special characters.
- Keep **Provisioned throughput** as the Capacity mode

- After creating it, you need to wait for a few minutes until it finishes Deployment.
    - We are using it as a NoSQL document database. Entries in the database are JSON documents.
- Go to the database resource and IGNORE the initial tutorial.
- Instead, go to Overview and click on the "+ Add Container" to create a new database
    - Database id: messagesdb
    - Container id: messages
    - Partition key: keep `/id`
    - Click OK.
- You can browse the created database and messages table, but it will be empty for now.
    - Once we create the second serverless function in the next exercise, it will start inserting messages into this table.

Exercise 4.3. Creating a Cloud Function for posting documents to the database

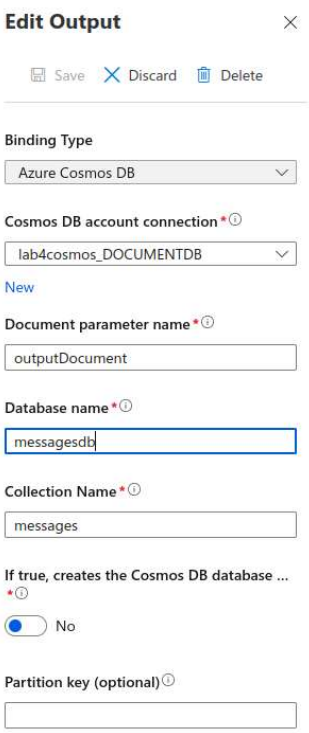
In this exercise, we will create a Cloud Function for entering messages into the Cosmos database.

- Create a new HTTP-triggered python function.
- Function name MUST be: **handleMessage**
- Make sure it is of **Anonymous** type (no API key or authentication needed)

Before we start modifying the code, let's set up the integration with the Cosmos DB database that we have created in the previous exercise. Azure function integrations control where the input of a Cloud function comes from and where the output of the function is delivered.

- Each function can have one trigger that defines the event that initiated a function and function can access the event parameters.
    - Function triggering Event acts as one of the inputs to the function.
    - In our case, HTTP POST or GET type requests act as the trigger for the function.
- Function can have zero or more additional inputs, for example to fetch a document of a file from a cloud storage
- Function should have one or more outputs (but can technically have zero), which allows the function to deliver or store output in multiple externals services.

- Open the integrations, and add a new Output connection/integration
    - Type of the output Should be Azure Cosmos DB.
    - Make sure, that an account connection to your previously created Cosmos DB is set up.
    - Document parameter name should stay: outputDocument
        - We will use this variable name in the code.
    - Database name should match the database id you created in the Cosmos DB: messagesdb
    - Collection Name should match the Cosmos DB container id: messages



The Integration page of the handleMessage Function should look something like this:

Let's now modify the code to process the entered message (submitted through the HTML form) and send it to the Cosmos DB.

- We need to fetch the form parameters (`msg`), calculate the current time, create a JSON object, and write it to the output stream of the outputTable object that we just set up in the Function integration output CosmosDB object.
- We also need to modify the function's Python `def main(...)` method arguments.
  - `def main(...)` arguments should include a variable for EVERY input and output stream of the function.
  - In the previous **htmlForm** function, we had one HTTP request input (`req: func.HttpRequest`) and the method returned HTTP response (`func.HttpResponse`)
    - It looked something like this: `def main(req: func.HttpRequest) -> func.HttpResponse:`
  - For this function, we need to add outputStream object with the name **outputDocument** (Name of this variable was configured in the Integration step) into the method arguments, modifying it into the following line:
    - `def main(req: func.HttpRequest, outputDocument: func.Out[func.Document]) -> func.HttpResponse:`
      - `req` object is the input
      - function returns an object of `func.HttpResponse` type
      - But now there is another output object called `outputDocument` (of type `func.Document`), which defines the content of the database record that can be created by this function.
  - After modifying the main method, let's access the message entered by the HTML form:
    - `msg = req.params.get('msg')`
  - We also need to generate a new ID for the document:
    - `rowKey = str(uuid.uuid4())`
      - Also, make sure to import the uuid library at the start of the program: `import uuid`
  - Then we can generate a JSON message to be entered into the DB. It should contain message content, the current timestamp, and an id.
    - make sure to also import the JSON library: `import json`
    - make sure to also import the datetime library: `from datetime import datetime`

```
json_message = {
    'content': msg,
    'id':  rowKey,
    'message_time': datetime.now().isoformat(" ", "seconds")
}
```

- To enter the JSON document into the database, we should set it as a value of the outputDocument object
  - Because it is func.Document type, we should use the `from_dict(...)` method, like this:
    - `outputDocument.set(func.Document.from_dict(json_message))`

- We should also return a message to the user that entering the message succeeded

```
return func.HttpResponse(
        f"Entered message was: {msg}",
        status_code=200,
        mimetype="text/html"
)
```

- To be nice, You can (but it is not required) add an HTML link back to the main page of the application.
    - Link it to the "/api/htmlForm" address of the first function.

Save and test the function.

Debugging and Monitoring Function executions

If you test manually, you need to add extra parameters (msg) and test it using POST request. Or you can try to use the submit function from the address of the first function.

- In case of errors:
    - Check the execution logs on the Test/Run page (Limited error logs).
    - Open the Monitor (Left side of the panel) page of a function, and check both the:
        - **Invocations** page - it may take some time (5 min) until function invocation results are shown
            - This page will usually show the most technical, code-level debugging messages and errors (if any)
        - Logs - You will need to keep this page running for a while to see any new logs of recent function invocations.

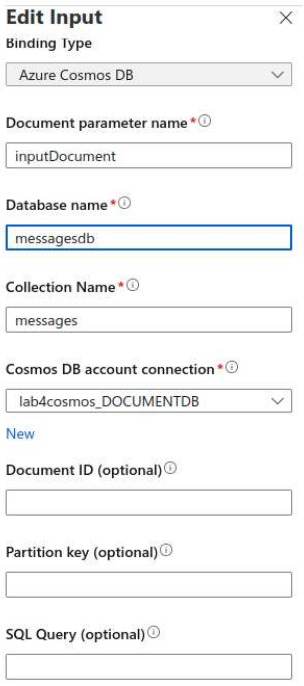
Exercise 4.4. Modifying the first htmlForm function to list previous messages from the database

Let's now modify the first function to also display the latest messages on the Web page. We need to modify two things:

1. Add Cosmos DB input integration to fetch data from the database messages table/container
2. Modify the code to add the message information into the resulting HTML.

Add Cosmos DB input integration to fetch data from the database messages table/container

- Go to htmlForm function Integration and add a new Input of type Azure Cosmos DB.
    - This will enable our function to access a list of the latest documents for the database.
    - All the parameters should be the same as for the second function Azure CosmosDB outputs
    - Leave Document ID, Partition key, SQL Query empty.
    - The name of the parameter for accessing the CosmosDB data from inside the function code is: **inputDocument**



The Integration page of the htmlForm Function should look something like this:

Let's now modify the code to add the message information into the resulting HTML.

- Add a new input **inputDocument** to the definition of the main function:

```
def main(req: func.HttpRequest, inputDocument) -> func.HttpResponse:
```

- inputDocument is (actually) a list of documents, and can be used as a list of Python dictionaries, where each JSON-like dictionary object is one message from the database
- For example:
  - `for doc in inputDocument:` would allow us to loop over all the documents in the database (limited to 50).
  - Accessing the individual document parameters (e.g., **content** and **message_time**) can be done like this: `doc['content']`
- Modify how the HTML is defined inside the function, to generate an HTML document something like this (where the messages are dynamically fetched from the database) :

HTTP response content

```
<title>Message board</title>
<body>
  <h4> Messages so far:</h4>
  <ul>
          <li>"test" <small>Posted on 2023-02-26 10:10:38</small></li>
            <li>"HI!" <small>Posted on 2023-02-26 10:14:50</small></li>
            <li>"test" <small>Posted on 2023-02-26 10:34:14</small></li>
            <li>"test" <small>Posted on 2023-02-26 10:36:31</small></li>
            <li>"shivananda was here" <small>Posted on 2023-02-26
11:18:51</small></li>
            <li>"None" <small>Posted on 2023-02-27 14:34:04</small></li>
        </ul>
  <h4> Enter a new message</h4>
  <form action="/api/handleMessage">
    <label >Your message:</label><br>
    <input type="text" name="msg"><br>
    <input type="submit" value="Submit">
  </form>
</body>
</html>
```

- You'll need to change how the `html_data` string object is created, as part of it needs to be generated based on the documents from the database
  - You can divide it into three parts:
    1. start of the HTML document, which contains the initial page header and title. And also the first list starting tag `<ul>`
    2. Dynamic content, which is generated based on messages from the database
       - You can append additional string content into the original by using something like `html_data += """ ... """`
    3. end of the HTML document, which contains the closing HTML `</ul>` tag, the HTML form and the closing HTML tags

Bonus task: Creating a Cloud Function for sending SMS messages every time a new message is entered into the database.

NB! It is strongly suggested that you try this task. It is not very difficult and illustrates the event Driven execution of Serverless Cloud Functions.

Cloud Functions can also be used to automate tasks. Your task is to create a Cloud Function that is automatically executed for every new document added to the database and which sends a new message alert to your phone as an SMS message.

Create a new Python Cloud Function.

- It should use the Cosmos DB trigger (not HTTP trigger)
  - It will be triggered every time there is a new message added to the messages table.
- Function type should NOT be Anonymous
  - We do not want users to be able to trigger it without credentials

Create a Twilio Free trial account and set up Twilo credentials.

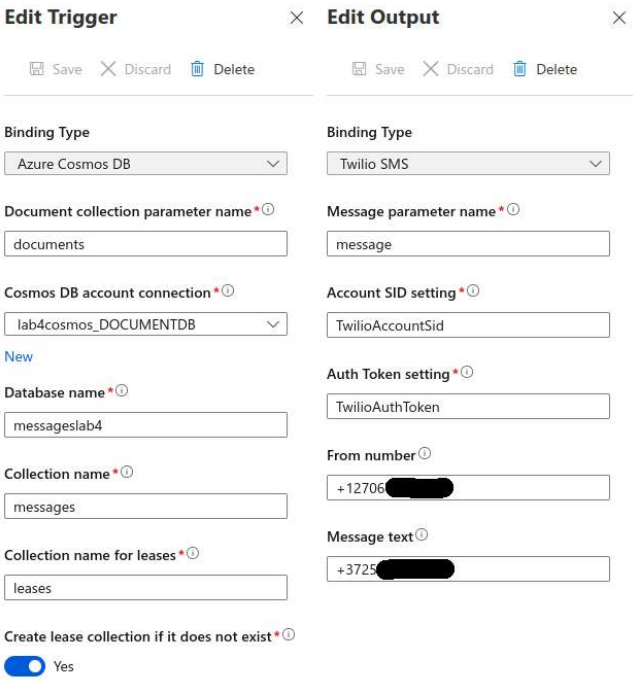
- Sign up for a free Twilio trial.
  - You need a mobile phone number for this.
  - Validate your mobile number
  - Set up your account so that you get access to a Twilio virtual mobile number, user ID, and Authentication Token.
    - These will be needed to configure the integration on the Azure side.

Configure the Twilio integration for the Azure cloud function.

- Change the output of the Function to be of Twilio SMS type.

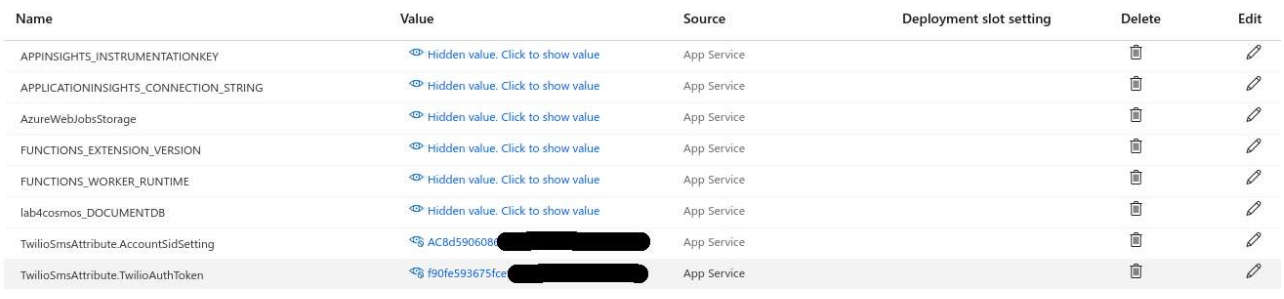
Configure both the input Azure Cosmos DB Trigger and the Twilio SMS outputs:

- Cosmos DB-related parameters should be the same as for the last function ones.
  - You will need to create another object called leases. Leave all leases-related parameters as default.
- For the Twilio binding:
  - set the "From" number to be your Twilio account virtual phone number.
  - set the "To" number to be your personal phone number that you added and validated in Twilio and which will receive the SMS message alerts.
  - Do not change AccountSidSetting and TwilioAuthToken values, these are the names of the Function App setting variables, and we define their values separately.



Now configure the Twilio service-related secrets.

- Go to Function App -> Settings and add two new Application settings:
  1. TwilioSmsAttribute.AccountSidSetting
     - Value should be the Twilio account id
  2. TwilioSmsAttribute.TwilioAuthToken
     - Value should be the Twilio account Authentication token



Updating the function code

- Check this example, on how to prepare a JSON message for the Twilio output integration: https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-twilio?tabs=in-process%2Cfunctionsv2&pivots=programming-language-python
- But be careful - Do not Follow the example blindly! - our function input (in the main method arguments) is different from the example code.
  - We are using the CosmosDB trigger, which means that the input is: `documents: func.DocumentList`
    - Input is a list of JSON documents (usually this list contains only one, the latest document)
    - You can use the same approach you used in the htmlForm function when accessing document data from the database.

Testing the trigger function

- Add new messages to the database, using the message board page.
  - Verify that the messages are added.
  - Check the trigger function invocations and logs
  - Check that SMS arrives at your phone

Bonus task Deliverables:

- Take a screenshot of the Invocation page (under the Monitor page of your function) of your Twilio trigger function, which shows the content of a successfully triggered Invocation
  - Take a LARGE screenshot so it also displays your username/email at the top right corner of the page.
- Source code of the function.

Deliverables

- Source code of all your Cloud Functions.
  - You can actually find the textual source code files of your functions inside the Storage Account (That you created or selected when creating the Function App)
    - Go to: Storage account -> Storage Browser -> File Shares -> name_of_the_dunction_app -> site -> wwwroot -> name_of_the_function
  - Provide the URL to the Cloud Function web endpoint, which displays the HTML form.
- Screenshot of your Cosmos DB database view, which should display one of the open documents, and show its content (with created_time and word_count).
  - Take a LARGE screenshot so it also displays your username/email at the top right corner of the page.
- Screenshots of your functions Monitoring page that displays at least one successful invocation. Make sure one of the Invocation are opened (showing the Invocation Details page of a successful invocation)
  - Take a LARGE screenshot so it also displays your username/email at the top right corner of the page.

Task	Lab 4 - FaaS
Current submission	ZIP (12:52 06.03.2023)
	If your homework consists of multiple files (for example HTML + CSS + JS) it should be archived before submitting.
File	Choose File No file chosen
Comment	

Submit

In case of issues, check these potential solutions to common issues:

- If some of the changes made in integrations or inside the code are not saved properly, or get overwritten for unknown reason:
  - Make sure you do not have multiple browser windows open for editing the same function code/integrations.
- If you get an error about 404 "Not Found" when trying to execute your function
  - Check Function integrations, if you have accidentally created two outputs or removed input.

The courses of the Institute of Computer Science are supported by following programs:


European Union
European Social Fund
Investing
in your future