Deliverable Session 05 - Information and Communication Technology

Rafael Antonio Echevarria Silva

In this session, we'll introduce you to the creation and management of a web service using the REST model using the Flask library. We'll implement a REST server capable of reading sensors, painting on the LED matrix, handling errors, and connecting to a database.

Ex. 1: Create a web service with Flask

In this first exercise, we'll configure a local REST server and then make it accessible over the network. Therefore, we first need to install the Flask library through the terminal. The next steps are to create a Python script with a simple path and run the server (locally). To provide external access, we need to run the following command in the terminal:

```
export FLASK_APP=name_of_script.py flask run -- host=´0.0.0.0´
```

Ex. 2: Read sensors on demand

This exercise tells us that we need to read data from SenseHat sensors and return it to the web service via REST endpoints. To do this, we create a new route with **/temperature**, read the sensor using SenseHat, and return the data. We also need to incorporate the date and time the sample was taken, so we need the **Datetime** library. We return both data to the web service. The way we return the data can cause problems, as it can be complex data. This is why it requires preprocessing or conversion to standard JSON, as it returns more complex structures (with nested dictionaries).

Ex. 3: Write data into sense HAT

In order to receive information from the user in the web service, we create a different route (/screen/pixel) to modify the LED matrix using HTTP parameters. To collect the parameters, we use the **request.args.get()** function to obtain the coordinates (x, y) and the color we want to include. We can see what the URL would look like:

[http://<your_IP_address>/screen/pixel?x=2&y=4&color=(255,255,0)]

Ex. 4: Error handling

Here we seek to detect errors that occur during execution and return appropriate HTTP responses. To do this, we need to activate debug mode with the terminal function:

```
export FLASK_DEBUG=1
```

With this, we can review the most common HTTP codes we may encounter, such as 404, 500, etc. Once we know what error the web service is executing, we can implement error handling in the script with Flask:

python @app.errorHandler(404) def not_found(e): return jsonify(error=str(e)), 404

Ex. 5: Access database

In this exercise, we must connect our REST server to a SQLite database to access the recorded information. The idea is to create **new RESTful endpoints** that allow us to query two key entities: **variables** and **sensors**. To do this, we use the **sqlite3** library, which allows us to run SQL queries from Python.

We design the routes (`/database/<name>`), where `<name>` will give us the parameter to run queries against the corresponding tables. When accessing these routes, the server will execute a query (**SELECT * FROM `<name>`**) and return the data in **JSON** format, using Flask's jsonify() function.

Ex. 6: Filters

This last exercise focuses on refining database queries by allowing date filters to be applied. The goal is to make requests to retrieve only measurements obtained within a certain time range. We use **GET parameters (from and to)** and use them in an SQL query with a clause like the following. Because we are saving the date data in the database as text, the clause would look like this:

```
SELECT * FROM measures WHERE date like "{from_date}" OR date like "{to_date}";
```

[http://127.0.0.1:5000/database/Filtro?from=2025-05-20%2017:50%&to=2025-05-20%2017:55%]