EXERCISE ONE)

For the first exercise, we decided on using the **POST** HTTP command to add models to the model registry, and the **GET** HTTP command to request a list of the models currently available in the registry and to start the humidity and temperature prediction process. For adding, we needed to pass the model in the body, and the only commands which allow us to always write the body are POST and PUT. We decided to go with post method since it is designed to send data to the service from an HTTP client. Specifically, we can send the model that we want to add to the Raspberry Pi.

As for requesting the models, there isn’t a need for writing a body so it’s logical to use GET. Also For the prediction part, we chose get. We don’t need to send any data on the body for the prediction, we are only sending 3 parameters, so it is better to use the get method. But for the alerts which are related to humidity and temperature thresholds, we used the **MQTT** protocol. The main motivation behind this decision was because we had to support sending an alert to multiple clients in the case of an absolute error that exceeds the defined thresholds for temperature and humidity and also if we used a Rest method, we needed to interrupt the prediction process every time we wanted to send an alert. MQTT is based on the publish/subscribe communication paradigm which means that we don’t send messages to specific devices based on their IP address; Instead, we publish our message on a topic and the broker allows all the subscribers to receive the message. Hence, we found MQTT as the more suitable protocol to use for this situation, not REST.

Furthermore, we dealt with possible errors that could arise when invoking the web service (wrong path, wrong parameters, checking for the existence of files, number of parameters, …)

EXERCISE TWO)

Regarding the second exercise, we decided on using the **MQTT protocol** and not REST. The main motivation behind our decision lies in the fact that when dealing with collaborative inference, we would like efficient communication through the network since we may depend on the service multiple times. MQTT is a lightweight protocol because all its messages have a small code footprint, and it is more protocol efficient, power efficient, and allows QoS (quality of service). Hence, it was the better option in this situation. The difference between both the fast and slow method in our case lies in the preprocessing taken. For the slow method, where the device was our notebook, we utilized the same parameters as those that the model was trained on. As for the fast method, we used:

**Sampling frequency=16kHz, resolution=16-bit, frame length=64ms, frame step=19.375ms, #Mel bins=32, lower frequency=20Hz, upper frequency=4kHz, #MFCCs=10.**

Of course, we implemented a success checker to determine when we should rely on the slow service. We tried monitoring the **Top-1 SoftMax** prediction and setting a threshold to determine whether the fast pipeline was confident or not. We also tried finding the difference between the top two SoftMax predictions and making sure that they are a certain threshold far away from each other. In the end, we used the Top-1 success checker method with a **threshold of 59%.**

With this configuration, we obtained an **accuracy of 91.25%, a communication cost of 1.882 MBs, and a latency of 37.698 ms.**