World Weather Measurement

Description

The goal of this system is to simulate weather measurement systems in cities all over the world. Because of missing real measurement devices, the values are taken from the API offered by Open Weather Map (OWM). These values are downloaded from OWM and pushed to the Context Broker via MQTT in a certain frequency. Furthermore, a watchdog and a graphical user interface (GUI) have been implemented. The watchdog helps to check if every city and every sensor monitored is being updated regularly and pushes errors to the Context Broker in case of failure. With the GUI the sensor values of the selected city are displayed. All of these software components are implemented in python. Additionally, the GUI uses PyQT.

Context Model

The Context Broker contains two kinds of entities, the `Weather_Measuring_Station` and the `Watchdog`.

The `Weather_Measuring_Station` is designed scalable and it is designed to add active attributes and commands for all devices with only changing the configuration and running some scripts. In the example configuration, it contains the active attributes `Temperature`, `Windspeed`, `Humidity` and `Pressure` and the commands `CpuLoad` and `UpdateCount`. Changing the static attributes requires a more serious change in the scripts. By default it contains a `Name` of the city as a string, a `Countrycode` for the country containing the city and the `Location` as a geo:point. Because of being a simple measurement device and being scalable, compounds aren't used and the structure is flat. Lazy attributes are not used because the current IoT Agent doesn't support this feature yet.

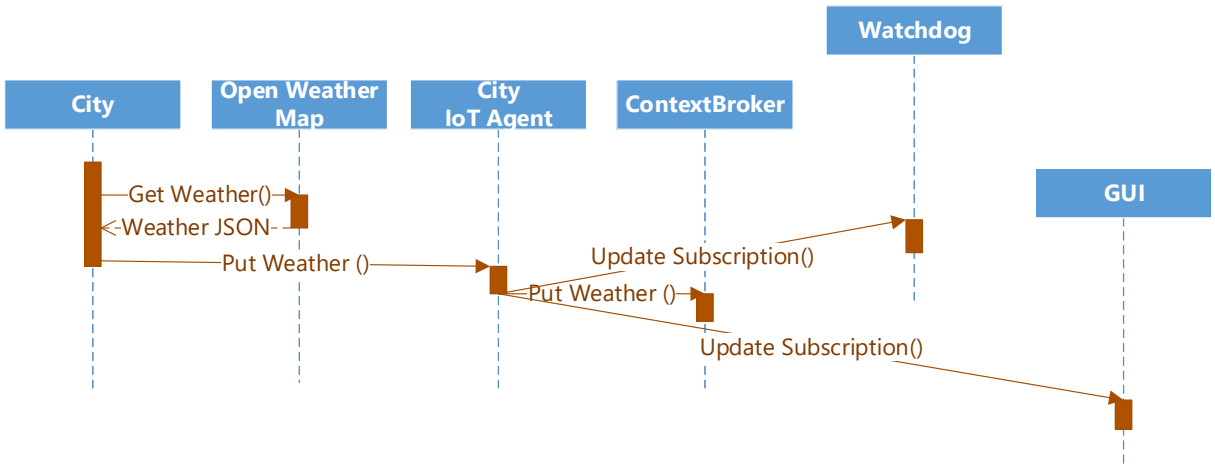
Active Attributes	Scalable (`Temperature, Windspeed, Humidity, Pressure, …`)
Lazy Attributes	-
Commands	Scalable (`CpuLoad, UpdateCount, …`)
Static Attributes	`Name, Countrycode, Location`

The `Watchdog` is a system to monitor the functionality of the cities. It contains two active attributes, the `CityErrors` and the `SensorErrors` to show if whole cities or only several sensors of a city have not been updated for a while. Therefor it contains a list of city names or sensor and city names which do not work properly. The watchdog is designed to run continuous even if a new measurement device in a new city is added. The commands `AddCity` and `RemoveCity` require the MQTT name of the city to be monitored or to be stopped to be monitored.

Active Attributes	`CityErrors, SensorErrors`
Lazy Attributes	-
Commands	`AddCity, RemoveCity`
Static Attributes	-

Communication Model

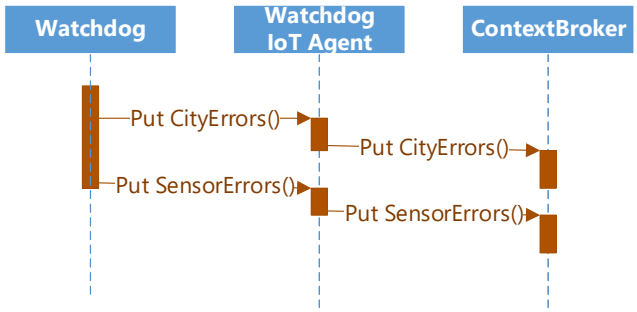
Weather Update of a City



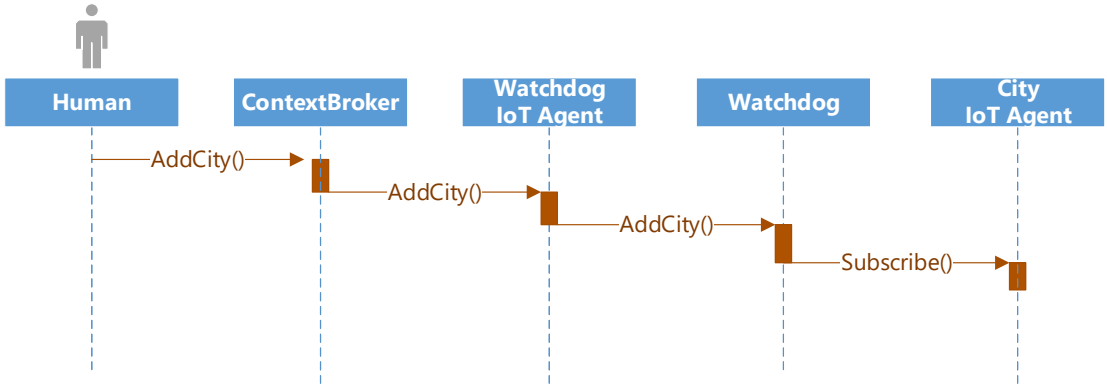
The standard use case is shown in this communication diagram. When a city is starting to update, it first gets new weather information from OWM and extracts and forwards the desired attributes to the IoT Agent. The IoT Agent puts these values to the Context Broker and notifies every component which subscribes to this device. In this application, the Watchdog and the GUI are possible subscribers.

Error Update of the Watchdog

If the Watchdog recognizes a lack of update, it pushes the information to its entity.



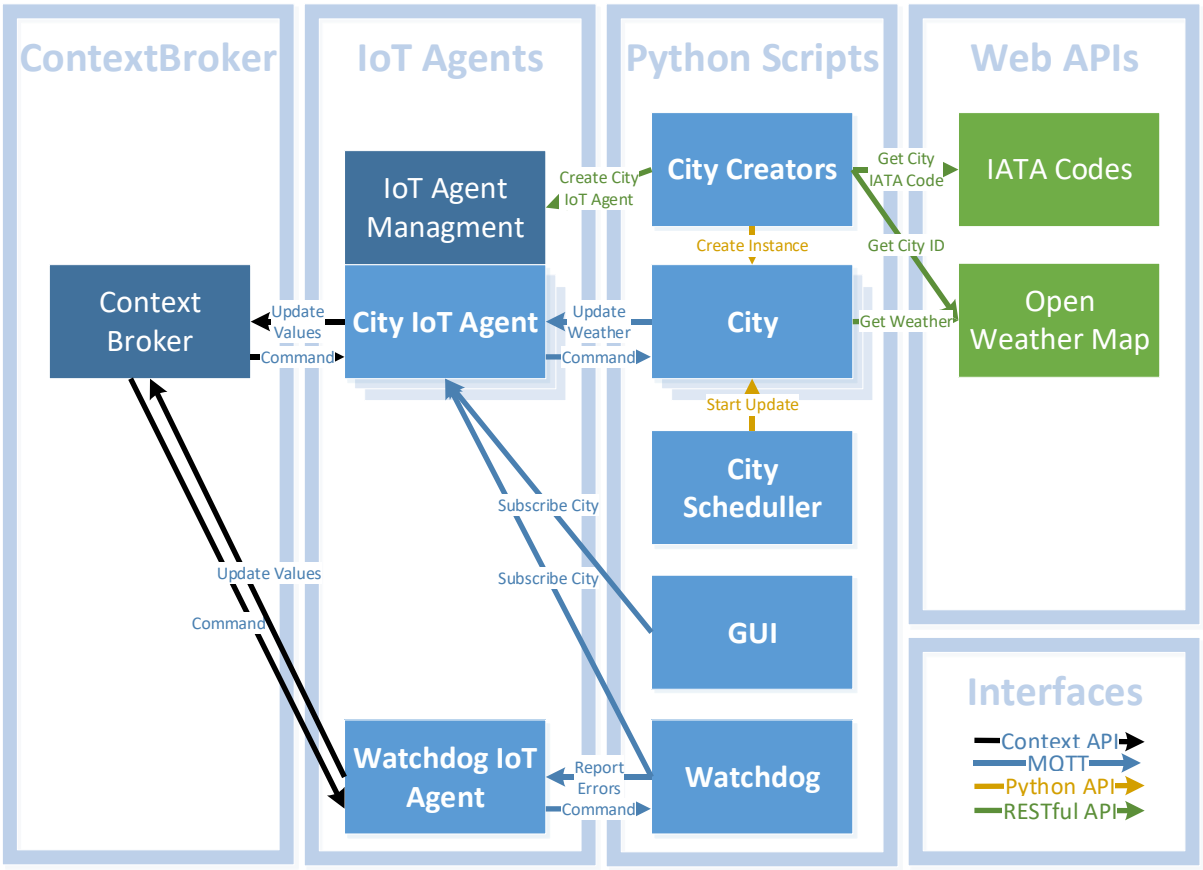
Add a new City to the Watchdog



If a new city should be added to the Watchdog, the command `AddCity` must be run and the name of the MQTT name of the city must be handed over. Finally, the Watchdog subscribes the active attributes of this city. `RemoveCity` works equally but unsubscribes the City IoT Agent.

Architecture

The system consists of four different types of elements. The Context Broker is the central part of this system and stores the data (in Orion). IoT Agents manage the communication with the IoT devices. Python scripts simulate the IoT devices. Furthermore, the Web APIs provide the value for the simulation of sensors and help to make the number of cities easily scalable.



Simulation of the Cities

City

The python script `city.py` is the main part of the simulator of a Weather Measuring Station. One instance contains the name of the belonging IoT Agent and the ID of the city in Open Weather Map. For simplification only cities with an IATA code are possible. This IATA code is contained in the name of the entity in the Context Broker and in the name of the IoT Agent. The city is only aware of the name of its IoT Agent and cannot communicate to the Context Broker directly.

Dependencies	
Fiware APIs	IoT Agent of type `Weather_Measuring_Station`
Python scripts	`city.py, city_definitions.py, city_commands.py`
Web APIs	Open Weather Map

City definitions and commands

The scripts `city_definitons.py` and `city_commands.py` aren't contained in the architecture diagram above. The `city_definitions.py` contains parameters concerning one or more python scripts. There are for example communication parameters like IPs and ports for different protocols, as well as the active attributes and commands used in the simulation of the city. The script `city_commands.py` contains methods intended to map commands of a city. Therefor these methods must have a certain interface.

City creators

The script `city_creators.py` helps to create not only city objects, but if desired to create the IoT Agents of cities as well. It is able to compose the name of the city in MQTT and get the ID of the city in Open Weather Map. Both are needed to generate city objects. Therefor it requires the APIs of the Open Weather Map and IATA Codes. Furthermore `city_creators.py` helps to generate a new IoT Agent or replace an old one and its entity in Context Broker. This is done via the Management API of the IoT Agents. Because of these capabilities, it is very easy, to create a huge number of cities by only knowing their names.

Dependencies	
Fiware APIs	Management API of IoT Agents, API of Context Broker
Python scripts	`city_creators.py, city.py, city_definitions.py`
Web APIs	Open Weather Map, IATA Codes

City scheduler

The script `city_scheduller.py` helps to simulate several cities on only one device. Therefor several cities are used and updates of all cities on one device are started in certain time intervals.

Watchdog

The script `watchdog.py` should monitor cities for pushing measurement updates within a certain interval. It is designed to run continuous even if a measurement device in a new city is added. The commands `AddCity` and `RemoveCity` enable adding new cities to and removing old ones from the subscription in MQTT and the monitoring. Detected errors are pushed to the Context Broker.

Dependencies	
Fiware APIs	IoT Agents of both type `Weather_Measuring_Station` and `Watchdog`
Python scripts	`city_definitions.py`

GUI

The script `context_consumer.py` acts as a Context Consumer by subscribing to the city data via MQTT and displaying the obtained data in a GUI. The desired city can be selected by a dropdown menu and as soon as new weather data of the selected city is pushed, the updated weather information is displayed in the GUI. The design of the GUI is stored in the script `GUI.ui` and was created with the Qt Designer, whereas the functionality of the GUI is implemented in Python using the PyQt library.

Dependencies	
Fiware APIs	IoT Agent of type `Weather_Measuring_Station`
Python scripts	`city_definitions.py`

Raspberry Pi

In order to push updates continuously, the scripts `city_scheduller.py` and `city.py` are designed to run on a raspberry pi. The python library logging is used to redirect the output to files and to select the types of output to be stored in these files. Furthermore, the script is configured to start automatically while booting. Therefore, the following line is added to `/etc/rc.local`

```
(sleep 10;python3 /home/pi/Documents/PythonScriptsIOT/main.py)&
```