TWIN4BUILD API DOCUMENTATION

TABLE OF CONTENT

Table of Contents

1. SCOPE ----------------------------------------------------------------------------------------- 2
2. DATA\_LAYER
   1. keycloak\_token.py ------------------------------------------------------------------- 3
   2. ql\_data\_fetch.py ---------------------------------------------------------------------- 6
3. DATABASE
   1. db\_data\_handler.py ------------------------------------------------------------------10
4. ML\_LAYER
   1. input\_data.py -------------------------------------------------------------------------13
   2. monitor\_api.py -----------------------------------------------------------------------16
   3. request\_to\_api.py --------------------------------------------------------------------20
   4. simulation\_api.py --------------------------------------------------------------------22

SCOPE

The scope of the Twin4build API is to run the simulation API using **‘smulation.py’** which is running on FastAPI.

Getting the json format of the inputs using the ‘**input\_data.py’** script and sending the data to the simulation API in the form of POST request.

From the API we receive simulation data which is then converted into desired format using ‘**transform’** function and is stored in the PostgreSQL database using the **‘db\_data\_handler.py’** script.

Keycloak is used for getting the access token from the Quantum Leap and QL Data Fetch is used for fetching the data from Quantum Leap.

**KeycloakAuthenticator Documentation**

**INTRODUCTION**

The KeycloakAuthenticator class is designed to handle authentication and token management with a Keycloak server. It provides methods for authenticating users, obtaining access tokens, refreshing tokens, and checking token expiration.

**Import Statements**

The code imports the required modules and classes for its functionality. It also imports a custom logger module (twin4build.logger.Logging) for logging information and errors.

**Class: KeycloakAuthenticator**

Constructor:

def \_\_init\_\_(self, server\_url, realm\_name, client\_id)

**Method: authenticate**

def authenticate(self, user, password) : Function Authenticate to Keycloak using user credentials.

**Method: get\_refresh\_token**

def get\_refresh\_token(self) -> str :

Returns -str: The refresh token associated with the authentication.

**Method: get\_access\_token**

def get\_access\_token(self) -> str

Returns - str: The access token associated with the authentication.

**Method: get\_refresh\_expiration\_time**

def get\_refresh\_expiration\_time(self)

Returns:- int: The time in seconds until the refresh token expires.

**Method: get\_access\_expiration\_time**

def get\_access\_expiration\_time(self): Get the expiration time of the access token.

Returns:- int: The time in seconds until the access token expires.

**Method: is\_access\_token\_expired**

def is\_access\_token\_expired(self) : Check if the access token has expired.

Returns:- bool: True if the access token has expired, False otherwise.

**Method: is\_refresh\_token\_expired**

def is\_refresh\_token\_expired(self) : Check if the refresh token has expired.

Returns:- bool: True if the refresh token has expired, False otherwise.

**Method: refresh\_token**

def refresh\_token(self) : This function refreshes the access token using the refresh token.

**Example Usage**

if \_\_name\_\_ == "\_\_main\_\_":

#Initialize the KeycloakAuthenticator with server URL, realm name, and client ID

authenticator = KeycloakAuthenticator(

server\_url="https://keycloak.example.com/auth",

realm\_name="your-realm-name",

client\_id="your-client-id"

)

# Authenticate with user credentials

authenticator.authenticate(user="your-username", password="your-password")

#Check if the access token has expired and refresh it if needed

if authenticator.is\_access\_token\_expired():

if not authenticator.is\_refresh\_token\_expired():

authenticator.refresh\_token()

#Get the refreshed access token

access\_token = authenticator.get\_access\_token()

#Print the access token

print("Access Token:", access\_token)

QL Data Fetch Documentation

INTRODUCTION

The APIClient class is designed to interact with the Quantum Leap API using Keycloak for authentication. It provides methods for fetching sensor data from the Quantum Leap API, handling token authentication, and working with time filters.

Constructor

def \_\_init\_\_(self) :

Initializes the APIClient class, reading configuration from configuration file and authenticating with Keycloak

Method: create\_url

def create\_url(self, room, sensor)

This function creates a URL for accessing the Quantum Leap API based on the provided room and sensor.

Parameters:

room (str): The room identifier.

sensor (str): The sensor identifier.

str: The generated API URL

Method: get\_configuration

def get\_configuration(self):

This function eads configuration data from a configuration file using the ConfigReader class

Method: create\_ql\_headers

def create\_ql\_headers(self):

Creates headers for interacting with the Quantum Leap API.

Returns:

dict: Headers for the API request

Method: get\_ql\_response

def get\_ql\_response(self, api\_url):

Fetches a response from the Quantum Leap API, ensuring token validity.

Parameters:

api\_url (str): The API URL to fetch data from.

Returns:

requests.Response: The API response

### Method: get\_ql\_sensor\_data

def get\_ql\_sensor\_data(self, api\_url)

Fetches sensor data from the Quantum Leap API using the provided API URL.

Parameters:

api\_url (str): The API URL to fetch data from.

Returns:

dict: The retrieved JSON response containing sensor data

Method: get\_all\_data

def get\_all\_data(self, room, sensor)

Fetches all sensor data for a given room and sensor using the Quantum Leap API.

Parameters:

room (str): The room identifier.

sensor (str): The sensor identifier.

Returns:

dict: The retrieved JSON response containing sensor data

Method: convert\_to\_custom\_format

def convert\_to\_custom\_format(self, input\_date\_time):

Converts a given date/time string to a custom format suitable for API queries.

Parameters:

input\_date\_time (str): The input date/time string.

Returns:

str: The formatted date/time string

### Method: get\_data\_using\_time\_filter

def get\_data\_using\_time\_filter(self, room, sensor, start\_time, end\_time):

Fetches sensor data from the Quantum Leap API based on a time filter.

Parameters:

room (str): The room identifier.

sensor (str): The sensor identifier.

start\_time (str): The start date/time for the time filter.

end\_time (str): The end date/time for the time filter.

Returns: dict: The retrieved JSON response containing sensor data

### Method: get\_latest\_n\_data

def get\_latest\_n\_data(self, room, sensor, n\_value):

Fetches the latest N values of sensor data from the Quantum Leap API.

Parameters:

room (str): The room identifier.

sensor (str): The sensor identifier.

n\_value (int): The number of latest values to fetch.

Returns:

dict: The retrieved JSON response containing sensor data

## **SQLAlchemy ORM Class Documentation**

INTRODUCTION

The provided code defines SQLAlchemy ORM classes to interact with PostgreSQL databases. These classes represent the tables ml\_inputs, ml\_inputs\_dmi, and ml\_simulation\_results. Below is the documentation for the provided classes, along with their methods and attributes.

Class: ml\_inputs

Represents the ml\_inputs table in the PostgreSQL database.

Attributes:

entity\_id: String. The entity identifier.

entity\_type: String. The entity type.

time\_index: DateTime. Time index of the entry.

\_\_original\_ngsi\_entity\_\_: JSON. Original NGSI entity data.

instanceid: String. Instance identifier.

datecreated: DateTime. Creation date.

datemodified: DateTime. Modification date.

iscontainedinbuildingspace: String. Containment status in building space.

co2concentration: Float. CO2 concentration value.

damper: Float. Damper value.

name: String. Name identifier.

opcuats: DateTime. OPC UA timestamp.

radiator: Float. Radiator value.

shadingposition: Float. Shading position value.

temperature: Float. Temperature value.

id: BIGINT. Primary key identifier.

### Class: ml\_simulation\_results

Represents the ‘ml\_simulation\_results’ table in the database.

### Class: ml\_inputs\_dmi

Represents the ‘ml\_inputs\_dmi’ table in the database.

### Class: db\_connector

A class that handles database connections and operations using SQLAlchemy ORM.

**Attributes**

* **engine**: SQLAlchemy Engine. The database connection engine.
* **session**: SQLAlchemy Session. The database session.

Methods

* **\_\_init\_\_(self)**: Initializes the **db\_connector** class.
* **get\_configuration(self)**: Reads configuration data from a configuration file.
* **get\_connection\_string(self)**: Generates a connection string for the database.
* **connect(self)**: Connects to the database using SQLAlchemy.
* **disconnect(self)**: Disconnects from the database.
* **create\_table(self)**: Creates the specified table.
* **add\_data(self, table\_name, inputs)**: Adds data to the specified table.
* **get\_all\_inputs(self, table\_name)**: Retrieves all data from the specified table.
* **get\_latest\_values(self, table\_name, roomname)**: Retrieves the latest data from the specified table.
* **get\_data\_using\_datetime(self, tablename, roomname, starttime, endtime)**: Retrieves data from the specified table within a time range.

Example Usage

if \_\_name\_\_ == "\_\_main\_\_":

connector = db\_connector()

connector.connect()

roomname = "O20-601b-2"

tablename = "ml\_simulation\_results"

inputs = {

# ... input data ...

}

connector.add\_data(table\_name=tablename, inputs=inputs)

queried\_data = connector.get\_latest\_values(table\_name=tablename, roomname=roomname)

start\_datetime = datetime.strptime("2023-08-17 08:50:00", '%Y-%m-%d %H:%M:%S')

end\_datetime = datetime.strptime("2023-08-22 10:40:00", '%Y-%m-%d %H:%M:%S')

queried\_data\_range = connector.get\_data\_using\_datetime(tablename=tablename, roomname=roomname, starttime=start\_datetime, endtime=end\_datetime)

connector.disconnect()

**Data Conversion Script Documentation**

INTRODUCTION

This script contains a class named **input\_data** that facilitates the conversion of data into a specific input format for simulation purposes. It handles retrieving data from a database, applying filters, and organizing it into the required input structure.

Import Statements:

- The script imports necessary modules such as os, sys, json, and datetime for data processing.

- It also imports custom modules related to database handling, configuration reading, and logging.

**Class: input\_data**

Constructor:

def \_\_init\_\_(self)

Initializes an instance of the input\_data class. It sets up configuration, database connection, and data processing.

Methods:

- get\_configuration(self)

Reads configuration settings from a configuration file using a ConfigReader.

Returns: None

- db\_connect(self)

Connects to the postgresql database using db\_connector.

Returns: None

- data\_from\_db(self, roomname, table\_names, data\_fetching\_method)

Retrieves data from the database using specified methods, such as fetching all inputs, data within a datetime range, or the latest values.

Parameters:

- roomname (str): The name of the room to fetch data for.

- table\_names (list of str): List of table names to retrieve data from.

- data\_fetching\_method (str): Method for fetching data from the database.

Returns: Dictionary containing retrieved data.

- get\_filter\_columns(self, table\_name)

Gets filter columns based on the table name by parsing the configuration settings.

Parameters:

- table\_name (str): Name of the table to get filter columns for.

Returns: List of filter columns.

- input\_data\_for\_simulation(self)

Processes input data for simulation. Reads JSON data, retrieves sensor data from the database, applies filters, and organizes the input data. Saves the data as a JSON file.

Returns: Dictionary containing processed input data.

- output\_data(self, response)

Modifies and returns the response dictionary by changing some information.

Parameters:

- response (dict): The original response dictionary.

Returns: Modified response dictionary.

Example Usage:

if \_\_name\_\_ == "\_\_main\_\_":

# Create an instance of the input\_data class

inputdata = input\_data()

# Process input data for simulation

input\_data\_dict = inputdata.input\_data\_for\_simulation()

# Output the processed input data

response = inputdata.output\_data(input\_data\_dict)

# Print the modified response

print("Modified Response:", response)

Import Statements:

- The script imports required modules for data processing, directory manipulation, and logging.

**Monitor Class Documentation**

Introduction

This script defines a Monitor class that is responsible for monitoring the performance of a building simulation. It utilizes data from a simulator and actual readings to plot and analyze performance metrics such as error, moving averages, anomaly signals, and more.

Import Statements:

- The script imports necessary modules for data processing, plotting, and performance analysis.

- It also imports custom modules related to simulation, plotting, and monitoring.

Class: Monitor

Constructor:

def \_\_init\_\_(self, model)

Initializes an instance of the Monitor class with the provided model. Sets up the simulator for simulation.

Methods:

- get\_ylabel(self, key)

Returns the appropriate y-label for a given property key.

Parameters:

* + - key (str): The key representing a property.

Returns: Y-label as a string.

- get\_error(self, key)

Computes and returns the absolute error between actual and simulated readings for a specific property key.

Parameters:

- key (str): The key representing a property.

Returns: Error values as a Pandas Series.

- get\_relative\_error(self, key)

Computes and returns the relative error (in percentage) between actual and simulated readings for a specific property key.

Parameters:

- key (str): The key representing a property.

Returns: Relative error values as a Pandas Series.

- get\_performance\_gap(self, key)

Computes and returns the performance gap (error) along with error bands for a specific property key.

Parameters:

- key (str): The key representing a property.

Returns: Error values, error band, and legend label.

- plot\_performance(self, save\_plots=False)

Plots the performance of simulated and actual readings, moving averages, and anomaly signals for each property.

Parameters:

- save\_plots (bool): Whether to save the plots as image files.

Returns: None

- save\_plots(self)

Saves the generated plots as image files.r

returns: None

- get\_moving\_average(self, x)

Computes and returns the moving average of the input data.

Parameters:

- x (Pandas Series): Input data for which to compute the moving average.

Returns: Moving average values as a Pandas Series.

- get\_MSE(self)

Computes and returns the Mean Squared Error (MSE) for each property.

Returns: Dictionary of MSE values for each property.

- get\_RMSE(self)

Computes and returns the Root Mean Squared Error (RMSE) for each property using computed MSE values.

Returns: Dictionary of RMSE values for each property.

- anomly\_function(self)

Placeholder function for anomaly detection code.

Returns: None

- get\_readings\_from\_db(self, startPeriod=None, endPeriod=None, do\_plot=False)

Placeholder function to retrieve simulation and actual readings from the database.

Parameters:

- startPeriod (datetime): Start time for data retrieval.

- endPeriod (datetime): End time for data retrieval.

- do\_plot (bool): Whether to plot the retrieved data.

Returns: None

- convert\_into\_json\_response(self)

Placeholder function to convert data matrices into a JSON response format.

Returns: None

- performance\_compare(self)

Main function to perform performance comparison between simulation and actual readings. Retrieves data, compares performance metrics, converts to JSON response.

Returns: None

Usage Example:

# Create an instance of the Monitor class with a simulation model

model = ... # Define your simulation model here

monitor = Monitor(model)

# Perform performance comparison and visualization

monitor.performance\_compare()

# Optionally, save generated plots as image files

monitor.save\_plots()

**Request To API Class Documentation**

Introduction

This Python script is designed to interact with a simulator API, retrieve simulation data, transform the data into suitable format, and store it in a database. The script can be used for data collection and storage purposes in a simulation environment.

Dependencies

Before using this script, we have to ensure that the following Python libraries are installed:

**os**: Operating system-specific functionality.

**sys**: System-specific parameters and functions.

**time**: Time-related functions.

**sched**: Event scheduling for time-based execution.

**json**: JSON (JavaScript Object Notation) encoding and decoding.

**requests**: HTTP library for making API requests.

**datetime**: Date and time manipulation.

**twin4build**: Custom libraries from the Twin4Build project (used for database and configuration handling).

Usage

The python code can be executed by running it as a standalone Python program. Here are the main components and their functions:

1. Imports and Testing

The script begins with importing necessary libraries and checks if it's being run as the main program. This section is primarily for testing purposes before packaging the code as a library or application.

2. Configuration

The ConfigReader class is used to read configuration settings from a file named conf.ini. The configuration includes parameters such as API URLs and database connection details.

3. Logging

The script initializes a logger to record events and errors during its execution. Logs are written to an 'ai\_logfile' for debugging and monitoring.

4. JSON File

The ‘create\_json\_file’ takes the object and the file path and create json file of the object using json dumps from the json python module

5. Convert Response

The ‘convert\_response\_to\_list’ function takes input response\_dict coming from the simulation API response running on FastAPI.

It iterates over the list of keys of dict and formats the data as desired to input to the database.

5. Validate Input Data

Validation of the input data is done using the table name “ ml\_inputs\_dmi “ observed column and the start\_time from the metadata of the input dict.

It returns true or false as validation of the input data

The format of the input data is as in the file :



5. Validate Response Data

Validation of the response data is done by checking the ‘time’ key in the response dict and the number of time input got from the response.

It returns true or false as validation of the response data

The format of the response data received from the simulation API is as :



6. transform\_dict Function

This function takes a dictionary of simulation data as input and transforms it into a specific format. It converts the timestamp format, extracts relevant data, and organizes it into a new dictionary for storage in the database.

7. request\_class Class

This class manages the process of interacting with the simulator API and storing the retrieved data in a database. Here's an overview of its methods:

**\_\_init\_\_(self):** Initializes the class by reading the configuration and connecting to the database.

**get\_configuration(self):** Reads the configuration settings from the conf.ini file.

**request\_to\_simulator\_api(self):** Sends a POST request to the simulator API which is running on FastAPI , retrieves simulation data, transforms it using the **transform\_dict** function, and stores it in the database.

It pulls the data from simulation with the start and end time, create json file ( for testing )

It valudated the input and response data else it disconnect from the database.

Transforms the response data into desired database format and stores the data into the database.

‘**getDateTine()’** function returns the start date and the end date based of 2 hours time gap based on Denmark Time.

Schduler is used to run the request api function to call , transform and store the data into the database every 2 hours.

6. Main Execution

The script's main execution section creates an instance of the request\_class class and calls the request\_to\_simulator\_api method to perform the data retrieval and storage operation

**Simulator API Documentation**

Introduction

This Python script serves as an API for running simulations using the Twin4Build project's components and models. It exposes an HTTP API using python’s **FastAPI** library that can receive simulation input data, perform the simulation, and return the simulation results.

Dependencies

Before using this script, we have to ensure that the following Python libraries and dependencies are installed:

**os**: Operating system-specific functionality.

**sys**: System-specific parameters and functions.

**datetime**: Date and time manipulation.

**pandas**: Data manipulation and analysis library.

**twin4build**: Custom libraries from the Twin4Build project (used for simulation and configuration handling).

**fastapi**: A modern web framework for building APIs with Python.

**uvicorn**: ASGI server that serves the FastAPI application.

**fastapi.middleware.cors:** Middleware for enabling Cross-Origin Resource Sharing (CORS).

Usage

The script can be executed by running it as a standalone Python program. Here are the main components and their functions:

1. Imports and Testing

The script begins with importing necessary libraries and checks if it's being run as the main program. This section is primarily for testing purposes before packaging the code as a library or application.

2. Configuration

The ConfigReader class is used to read configuration settings from a file named **conf.ini**. The configuration includes parameters such as the filename of the data model and simulation step size.

3. Logging

The script initializes a logger to record events and errors during its execution. Logs are written to an **'ai\_logfile**' for debugging and monitoring.

4. SimulatorAPI Class

This class defines the **FastAPI**-based API for running simulations. Here's an overview of its methods and functions:

**\_\_init\_\_(self):** Initializes the class. It sets up FastAPI, enables CORS, and defines the API route for simulation.

**get\_configuration(self)**: Reads the configuration settings from the conf.ini file.

**get\_simulation\_result(self, simulator):** Takes a simulator instance, retrieves simulation results, and organizes them into a dictionary format.

**run\_simulation(self, input\_dict: dict)**: The main simulation function. It loads a data model, performs the simulation, and returns the simulation results in dictionary format. If an error occurs during simulation, it returns an error message.

5. Main Execution

In the main section of the script, an instance of the SimulatorAPI class is created, and the FastAPI server is started on port 8005.

API Endpoint

The API endpoint for running simulations is defined as POST /simulate. It accepts a JSON payload with simulation input data and returns the simulation results in JSON format.

Conclusion

This Python script provides a web API for running simulations using the Twin4Build project's components and models. It enables the execution of simulations via HTTP requests, making it suitable for integration into larger systems or applications. The API can be accessed by sending a POST request to http://localhost:8005/simulate with the required input data in the request body.