Business Process Automation SS24 Documentation

version 1.0

June 18, 2024

Contents

BPA SS24 Code Dokumentation!

Authors: Konrad Adamski, Tom Bischopink, Jessica Knick, Jakob Laufer, Tom Marinovic, Fabian Kuhne

Introduction

Welcome to the documentation for BPA SS24. This documentation provides an overview of the code structure, usage examples, and detailed explanations of each module and function.

About This Project

The BPA SS24 code is a Python package that provides a set of tools for handling data from various sources, including OPCUA servers, MQTT brokers and AAS shells. The code is designed to be easy to use and flexible, allowing users to quickly and easily access and manipulate data from different sources.

Getting Started

Please install Python 3.9 or higher.

Create Virtual Environment

```
python3.9 -m venv .venv
```

Activate Virtual Environment

```
source .venv/bin/activate
```

Install requirements.txt

```
pip install -r requirements.txt
```

Add .env file

- create a **.env** file
- put the following credentials into the .env file

```
OPCUA_URL_MOCKUP=...
OPCUA_URL=...
AAS_URL=...
MQTT_URL=...
MQTT_PORT=...
```

Start the Flask-App

```
python3 app.py
```

Open the following URL: **http://127.0.0.1:3000**

Run the OPCUA Simulation Server

```
cd Simulation
```

```
python3 OPC_UA_SimServer.py
```

Code ===

src package

Subpackages

src.utils package

Submodules

src.utils.AASManager module

class src.utils.AASManager.**AASManager**(logger_on=True)

Bases: **object**

AASManager Class

The *AASManager* class handles communication with the Asset Administration Shell (AAS) registry to manage inspection plans and responses.

AAS_Registry_URL

URL of the AAS registry.

Type: str

ID

Key to identify the asset in the AAS.

Type: str

__init__(self)

Initializes the AASManager and logs the initialization.

get_inspection_plan(self, auto_id)

Retrieves the inspection plan for a given auto_id. :param auto_id: The ID of the auto for which to fetch the inspection plan. :return: The inspection plan or None if an error occurs.

get_inspection_response(self, auto_id)

Retrieves the inspection response for a given auto_id. :param auto_id: The ID of the auto for which to fetch the inspection response. :return: The inspection response or None if an error occurs.

put_inspection_response(self, auto_id, json_dict)

Submits the inspection response (in JSON format) for a given auto_id. :param auto_id: The ID of the auto for which to submit the inspection response. :param json_dict: The inspection response data in JSON format. :return: None.

_get_asset_href(self, data, id_short)

Finds the asset href in the provided data by the short identifier. :param data: JSON data from which to find the href. :param id_short: The short identifier for the asset. :return: The href if found, otherwise None.

_get_submodelIdentifier(self, aas_ip_port, idShort)

Fetches the submodel identifier for a given AAS. :param aas_ip_port: IP and port for AAS. :param idShort: Short identifier for the submodel. :return: Submodel identifier or None.

_get_attachment(self, ip_port, submodelIdentifier, idShortPath)

Fetches the attachment (inspection plan) for a given submodel identifier and path. :param ip_port: IP and port for AAS. :param submodelIdentifier: Identifier for the submodel. :param idShortPath: Path for the submodel element. :return: Inspection plan if found, otherwise None.

_put_attachment(self, ip_address, submodelIdentifier, idShortPath, ass_file_name, json_data)

Submits an attachment (inspection response) for a given submodel identifier and path. :param ip_address: IP address for AAS. :param submodelIdentifier: Identifier for the submodel. :param idShortPath: Path for the submodel element. :param ass_file_name: File name for the attachment. :param json_data: JSON data for the attachment. :return: None.

Usage:

aas_manager = AASManager() inspection_plan = aas_manager.get_inspection_plan(auto_id="some_auto_id")
aas_manager.put_inspection_response(auto_id="some_auto_id", json_dict={"key": "value"})

ID = *'idShort'*

get_all_idShorts ()
Extracts all 'idShort' values from the JSON data under the 'result' key. :return: A list of all 'idShort' values.

get_inspection_plan (auto_id)
Get inspection plan by auto_id. :param auto_id: The ID of the auto for which to fetch the inspection plan. :return: The inspection plan or the auto_id if an error occurs.

get_inspection_response (auto_id)
Get inspection response by auto_id. :param auto_id: :return:

put_inspection_response (auto_id, json_dict)
Put inspection response (JSON) by auto_id . :param auto_id: :param json_dict: :return:

requests = *<module 'requests' from '/Users/tom/Documents/AWI Msc./2. Semester/Business Process Automation/bpa-beleg-ss24/bpaenv/lib/python3.11/site-packages/requests/__init__.py'>*

test_connection ()

src.utils.Logger module

class src.utils.Logger.**SingletonLogger** (name='src.utils.Logger', log_file='app.log')
Bases: **object**
SingletonLogger Class
The *SingletonLogger* class provides a singleton implementation of a logger using Python's logging module. This ensures that only one instance of the logger is created, which can be used across different modules for consistent logging.

_instance
The singleton instance of the logger.

Type: SingletonLogger

name
The name of the logger.

Type: str

log_file
The file path for the log file.

Type: str

logger
The logger instance.

Type: logging.Logger

__new__ (cls, name=__name__, log_file='app.log')

Creates a new instance of SingletonLogger if it does not already exist, and sets up the logging handlers. :param name: The name of the logger. :param log_file: The file path for the log file. :return: The singleton logger instance.

_setup_handlers(`self`**)**
Sets up the logging handlers (file and console) and formatters for the logger.

Usage:

logger = SingletonLogger() logger.info("This is an info message")

src.utils.util_aas module

`src.utils.util_aas.`**encode_to_base64** (`original_string: str`)
Encodes a given string into its Base64 representation.
This function takes a string, encodes it to bytes using UTF-8 encoding, then converts those bytes into a Base64 encoded string. Base64 is commonly used when there is a need to encode binary data, especially when that data needs to be stored and transferred over media that are designed to deal with textual data.

Parameters: **original_string** (*str*) – The original string to be encoded into Base64.
Returns: A Base64 encoded string of the original string.
Return type: str

Example

encoded_string = encode_to_base64("Hello, World!") print(encoded_string) # Outputs: SGVsbG8sIFdvcmxkIQ==

src.utils.util_config_cars module

`src.utils.util_config_cars.`**get_auto_id** (`rfid`)
Retrieves the auto ID associated with a given RFID from the cars configuration.
This function searches through a preloaded cars configuration JSON to find the auto ID corresponding to the specified RFID code.

Parameters: **rfid** (*str*) – The RFID code to search for in the car configurations.
Returns: The auto ID if found; otherwise, None.
Return type: str or None

Example

auto_id = get_auto_id('12345RFIDCode')

`src.utils.util_config_cars.`**get_car_name** (`auto_id`)
Retrieves the car model name associated with a specific auto ID from the cars configuration.

Parameters: **auto_id** (*str*) – The auto ID to search for in the car configurations.
Returns: The name of the car model if found; otherwise, None.
Return type: str or None

Example

car_name = get_car_name('AU123ID')

`src.utils.util_config_cars.`**get_cars_json** ()
Retrieves the cars configuration data from a JSON file specified by the configuration path.
This function attempts to load a JSON file that contains configuration data for various car models, handling and logging any file not found errors.

Returns: A dictionary containing cars configuration data, or None if the file is not found.
Return type: dict or None

Example

cars_data = get_cars_json()

`src.utils.util_config_cars.`**get_rfid_forSimulation** (`auto_id`)
Retrieves the RFID for a specified auto ID for simulation purposes.

This function is primarily used in a simulated environment to fetch the RFID associated with an auto ID, formatted specifically for simulation outputs.

Parameters: **auto_id** (*str*) – The auto ID whose RFID is needed for simulation.
Returns: The formatted RFID string if found; otherwise, None.
Return type: str or None

Example

rfid_sim = get_rfid_forSimulation('AU123ID')

`src.utils.util_config_cars.`**`save_car_data`**(`data`)
Saves the provided cars configuration data back to the JSON file.

Parameters: **data** (*dict*) – The car configuration data to save.
Returns: True if data is successfully written; False otherwise due to an IO error.
Return type: bool

Example

success = save_car_data(modified_data)

`src.utils.util_config_cars.`**`set_car_rfid`**(`auto_id`, `new_rfid`)
Updates the RFID for a specified auto ID in the car configuration.

Parameters:
- **auto_id** (*str*) – The auto ID whose RFID needs to be updated.
- **new_rfid** (*str*) – The new RFID to assign to the auto ID.

Returns: True if the RFID is updated and saved successfully; False otherwise.
Return type: bool

Example

success = set_car_rfid('AU123ID', 'new12345RFIDCode')

`src.utils.util_config_cars.`**`update_car_data`**(`auto_id_list`)
Updates or adds new car data for the provided list of auto IDs in the cars configuration.

Parameters: **auto_id_list** (*list of str*) – List of auto IDs to update or add in the configuration.
Returns: True if the configuration was updated and saved successfully; None otherwise.
Return type: bool or None

Example

updated = update_car_data(['AU123ID', 'AU456ID'])

src.utils.util_inspection_response module

`src.utils.util_inspection_response.`**`create_response_plan`**(`inspection_plan`, `inspection_response_simplified`)
Generates a response plan based on an inspection plan and simplified inspection responses.
This function maps inspection responses to the relevant sections of an inspection plan, considering specific conditions and keys to update the response accordingly.

Parameters:
- **inspection_plan** (*dict*) – Dictionary containing details of the inspection plan.
- **inspection_response_simplified** (*dict*) – Simplified responses from the inspection camera.

Returns: A dictionary containing the updated response plan based on the provided inspection inputs.
Return type: dict

Example

response_plan = create_response_plan(inspection_details, simplified_responses)

`src.utils.util_inspection_response.`**`get_simplified_inspection_response`**(`data_in`, `schwellwert=0.6`)
Simplifies raw inspection data into a more manageable format based on a threshold confidence value.

This function processes raw detection data to evaluate which items meet specified confidence and condition thresholds. It uses internal functions to transform detections and generate a simplified response.

Parameters:
- **data_in** (*dict*) – Raw data containing detections from an inspection.
- **schwellwert** (*float, optional*) – The threshold for determining if a detection is significant, default is 0.6.

Returns: A dictionary with simplified inspection responses keyed by class name.

Return type: dict

Example

```
simplified_response = get_simplified_inspection_response(raw_data)
```

Submodules

src.MQTT_Camera module

class `src.MQTT_Camera.`**`MQTTClient`**(`broker_address_in='141.56.180.177'`, `port_in=1883`)

Bases: **`object`**

Manages communication with an MQTT broker, specifically for handling connection, messaging, and response processing related to camera inspection requests and results.

This class facilitates the publishing of inspection requests and the handling of asynchronous responses from an MQTT broker, employing events to manage the synchronous aspects of asynchronous communications.

`client`

Instance of the MQTT client using MQTT v5.0 protocol.

Type: mqtt.Client

`broker_address`

Address of the MQTT broker.

Type: str

`port`

Port number to connect to the MQTT broker.

Type: int

`request_topic`

MQTT topic for publishing inspection requests.

Type: str

`response_topic`

MQTT topic for subscribing to receive inspection responses.

Type: str

`response_payload`

Stores the latest received response payload.

Type: dict

`is_connected`

True if the client is successfully connected to the broker.

Type: bool

`connection_established`

An event to signal successful connection establishment.

Type: threading.Event

message_received
An event to signal the receipt of a new message.

Type: threading.Event

__init__ (self, broker_address_in, port_in)
Initializes the MQTTClient with specified broker address and port. Defaults are provided through module-level configuration.

on_connect (self, client, userdata, flags, reason_code, properties=None)
Handles successful connection events, subscribes to the response topic, and signals readiness.

on_disconnect (self, client, userdata, reason_code, properties)
Handles the disconnection event, resets connection flags, and logs the event.

on_message (self, client, userdata, msg)
Processes received messages, decodes JSON payloads, and logs the data.

test_connection (self)
Attempts to connect to the MQTT broker to check the connection status.

connect (self)
Establishes a connection with the MQTT broker and begins the network loop.

send_request (self, message)
Publishes a request message to the specified request topic.

request_response_cv (self, message, timeout)
Sends a request for camera inspection and waits for a response within the specified timeout.

disconnect (self)
Stops the network loop and disconnects from the MQTT broker.

Usage:

mqtt_client = MQTTClient() mqtt_client.connect() response = mqtt_client.request_response_cv() mqtt_client.disconnect()

connect ()

disconnect ()

on_connect (client, userdata, flags, reason_code, properties=None)

on_disconnect (client, userdata, flags, reason_code, properties)

on_message (client, userdata, msg: MQTTMessage)

request_response_cv (message='Triggering Camera', timeout=10)

send_request (message='Triggering Camera')

test_connection ()

src.OPC_UA_Subscriber_RFID_Reader module

class src.OPC_UA_Subscriber_RFID_Reader.**OPC_UA_Subscriber** (is_simulation=True)
Bases: **object**

A class to manage connections and data subscriptions to an OPC UA server, handling data changes and executing callbacks based on the RFID readings from a designated node. This class can operate in both simulation and production modes defined by its initialization parameter.

is_simulation
Determines if the subscriber will connect to a mock-up or real OPC UA server.

Type: bool

test_connection_successful
Indicates if the last connection test to the server was successful.

Type: bool

is_connected
True if the subscriber is currently connected to the server.

Type: bool

opcua_url
The URL to the OPC UA server, determined based on the mode of operation.

Type: str

ass_manager
An instance of the Asset Administration Shell Manager for handling data.

Type: AASManager

latest_auto_id_lock
A lock for thread-safe operations on the latest_auto_id.

Type: threading.Lock

latest_auto_id
The last read auto ID from the OPC UA server.

Type: str

client
An OPC UA client connected to the server.

Type: Client

sub
A subscription object for the OPC UA client.

Type: Subscription

handler
An inner class instance to handle data change notifications.

Type: SubHandler

hostname
Hostname parsed from the OPC UA URL.

Type: str

port
Port number parsed from the OPC UA URL.

Type: int

__init__ (is_simulation=True)
Initializes the subscriber, sets up the URL, and connects to the server.

test_connection (timeout=4)
Tests the connection to the OPC UA server with a specified timeout.

connect ()
Establishes a connection with the OPC UA server and subscribes to node changes.

disconnect ()
Disconnects from the OPC UA server and cleans up resources.

Inner Class:

SubHandler: Handles data change notifications from the OPC UA server, processes RFID data, triggers callbacks, and logs responses based on the inspection plan retrieved using the latest auto ID.

class **SubHandler** (outer)
Bases: **object**

datachange_notification (node, val, data)

register_callback (callback)

connect ()

disconnect ()

test_connection (timeout=4)

app module

app.**check_aas_connection_status** ()
Checks and returns the connection status of the Asset Administration Shell (AAS).

app.**check_connection** ()
Performs a connection test for the main inspection handler and returns the status.

app.**handle_switch** ()
Handles the switch between simulation and actual OPC UA server settings.

app.**index** ()
Serves the main page of the application, handles POST to update RFID settings, and dynamically updates car data from the AAS.

app.**inspection_plan** (auto_id)
Retrieves and displays the inspection plan for a specified auto ID.

app.**inspection_response** (auto_id)
Retrieves and displays the inspection response for a specified auto ID.

app.**log_content** ()
Fetches and returns the content of the log file.

app.**reset_logs** ()
Resets the log file to only keep the last few entries.

app.**start_inspection** ()
Initiates the inspection process if not already active.

app.**stop_inspection** ()
Stops the ongoing inspection process if currently active.

app.**view_logs** ()
Displays the logs page to view application logs.

InspectionHandler module

class InspectionHandler.**InspectionHandler**(is_simulation=True)

Bases: **object**

InspectionHandler Class

The *InspectionHandler* class is responsible for managing the inspection process, utilizing MQTT for communication with a camera system and OPC UA for subscribing to an assembly line. It integrates camera inspection responses with inspection plans to generate appropriate responses.

is_simulation

Indicates if the system is in simulation mode. Defaults to True.

Type: bool

opcua_subscriber

An instance of the OPC UA subscriber for the assembly line.

Type: OPC_UA_Subscriber

mqtt_client

An instance of the MQTT client for communication with the camera system.

Type: MQTTClient

runner_thread

Thread to run the main loop.

Type: threading.Thread

test_connection_successful

Indicates if the last connection test was successful.

Type: bool

is_connected

Indicates if the handler is connected to both OPC UA and MQTT.

Type: bool

stop_event

Event to signal stopping of the main loop.

Type: threading.Event

__init__(self, is_simulation=True)

Initializes the InspectionHandler with optional simulation mode.

test_connection(self)

Tests the connections to both the OPC UA subscriber and the MQTT client.

connect(self)

Connects the MQTT client and OPC UA subscriber to their respective services.

disconnect(self)

Disconnects the MQTT client and OPC UA subscriber from their respective services.

get_inspection_response(self, inspection_plan)

Requests a camera inspection response via MQTT, processes the response, and generates an inspection response plan.

run_loop(self)

Main loop that keeps testing the OPC UA connection and handles inspection process.

start (`self`)
Starts the inspection process by connecting to services and running the main loop.

stop (`self`)
Stops the inspection process by signaling the main loop to stop and disconnecting from services.

Usage:

handler = InspectionHandler() handler.start() handler.stop()

connect ()
Connects the MQTT client and OPC UA subscriber to their respective services, updating the is_connected attribute.

disconnect ()
Disconnects the MQTT client and OPC UA subscriber from their respective services, and updates the is_connected attribute.

get_inspection_response (`inspection_plan`)
Requests a camera inspection response via MQTT, processes the response, and generates an inspection response plan.

Parameters: **inspection_plan** (*dict*) – The inspection plan received from the OPC UA subscriber.
Returns: The generated inspection response plan.
Return type: dict

run_loop ()
Main loop that keeps testing the OPC UA connection and handles the inspection process, ensuring continuous operation.

start ()
Starts the inspection process by connecting to services and running the main loop.

Returns: Status message indicating success or failure to start.
Return type: str

stop ()
Stops the inspection process by signaling the main loop to stop and disconnecting from services.

Returns: Status message indicating the system is inactive.
Return type: str

test_connection ()
Tests the connections to both the OPC UA subscriber and the MQTT client, updating the test_connection_successful attribute.

Indices and tables

- **genindex**
- **modindex**
- **search**

Index

Python Module Index

a

i

s