The simulated I4.0 lab

Software Technology 4th semester project

Introduction **0**

Warehouse **2**

Asset communication 2

Features and functionality 3

State information 4

AGV **4**

Asset communication 5

Features and functionality 5

State information 7

Assembly station **8**

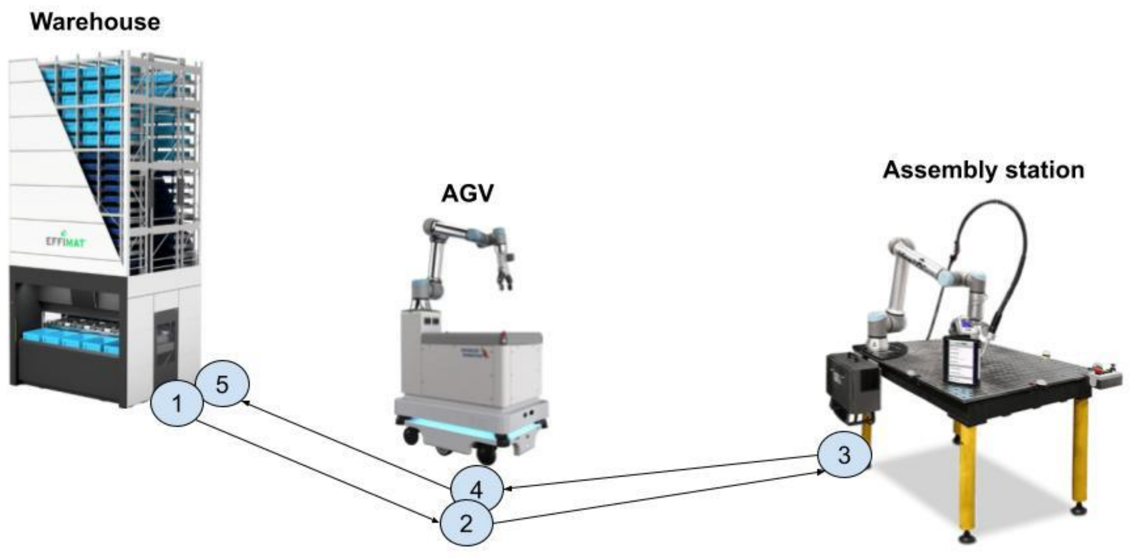
Asset communication 8

Features and functionality 9

State information 10

Introduction

This document will explain the features and functionality of the simulated lab assets that are involved as part of the (ST4-PRO) 4th semester Software Technology project, along with documentation explaining the implementation and use of the interfaces. Please refer to the project description for the project description and goals.



Warehouse

Asset communication

The warehouse in the SDU I4.0 lab lets users store parts and pieces of the current drone product line, while handling all of the internal shelf-mechanics and optimization algorithms internally on the PLC. It is typically operated by users through an onboard interface via a touch-screen mounted on the front of the machine. One of the goals of the Industry 4.0 initiative is easy connectivity to external systems using software, so the vendor's software has an implementation of a web service that communicates using SOAP as its messaging protocol.

> *“**SOAP** (Simple Object Access Protocol) is a messaging protocol specification for exchanging structured information in the implementation of web services in computer networks. It uses XML Information Set for its message format, and relies on application layer protocols, most often Hypertext Transfer Protocol (HTTP), although some legacy systems communicate over Simple Mail Transfer Protocol (SMTP), for message negotiation and transmission.*
>
> *SOAP allows developers to invoke processes running on disparate operating systems (such as Windows, macOS, and Linux) to authenticate, authorize, and*

communicate using Extensible Markup Language (XML). Since Web protocols like HTTP are installed and running on all operating systems, SOAP allows clients to invoke web services and receive responses independent of language and platforms."

- https://en.wikipedia.org/wiki/SOAP

To communicate with the service, you will have to implement a component that can communicate with the SOAP service. For most programming languages you can find public libraries that greatly simplify this process, so look at the options and choose a fitting one for your solution.

Features and functionality

The simulated warehouse has a few basic features which are implemented and exposed by the interface. These make it possible to do basic box retrieval and inserts, as well as checking the current inventory of the asset, which also returns state information.
The warehouse consists of removable boxes, which are placed in non-removable trays inside the warehouse itself. For interaction with the API, you will be working with tray IDs and content names for pick and insert operations.

The SOAP endpoint for implementation can be found at **http://localhost:8081/Service.asmx**, which contains the WSDL that describes the details of requests and data structure.

Method	**Description**
PickItem(int trayId)	Request an item from the warehouse.
InsertItem(int trayId, string name)	Insert an item into the warehouse.

GetInventory	`Return example:`
	`{`
	`    "Inventory":`
	`        [{`
	`            "1": "Item 1",`
	`            "2": "Item 2",`
	`            "3": "Item 3",`
	`            "4": "Assembly 1",`
	`            "5": ""`
	`        }],`
	`    "State": 0,`

	```
	    "TimeStamp": 12:34:56
	}
	```

State information

State	Description
0	Idle
1	Executing
2	Error

AGV

Asset communication

In SDU's I4.0 lab we use AGVs to move parts between stations where transportation is not easily accessible via other means (conveyor belt etc.). AGVs provide a flexible solution to automated transportation, since they can handle collision, pathing and optimization of those paths as they are repeated.

The current AGVs in the I4.0 lab communicate using a REST API, which is widely regarded as a flexible web service that is easy to work with. Often regarded as the successor to SOAP, as it provides similar functionality but is simpler to work with.

> *"Representational state transfer (**REST**) is a software architectural style that was created to guide the design and development of the architecture for the World Wide Web. REST defines a set of constraints for how the architecture of an Internet-scale distributed hypermedia system, such as the Web, should behave. The REST architectural style emphasizes the scalability of interactions between components, uniform interfaces, independent deployment of components, and the creation of a layered architecture to facilitate caching components to reduce user-perceived latency, enforce security, and encapsulate legacy systems.[1]*

REST has been employed throughout the software industry and is a widely accepted set of guidelines for creating stateless, reliable web APIs. A web API that obeys the REST constraints is informally described as RESTful. RESTful web APIs are typically loosely based on HTTP methods to access resources via URL-encoded parameters and the use of JSON or XML to transmit data."

- *https://en.wikipedia.org/wiki/Representational_state_transfer*

Features and functionality

The AGV consists of a set of pre-programmed paths and programs, which is programmed on the teach pendant (controller/screen) of the AGV. Through a REST web service it is possible to execute these predefined programs and request status-data.

The web service of the simulated asset (localhost for local execution) is hosted on **http://localhost:8082/v1/status/**.
Currently, the implementation requires a user to load a program with a PUT request and execute the preloaded program with another PUT request - similar to loading a gun and then pulling the trigger in two separate steps. The `"State": 1` attribute must be present when loading a new program, like in the example below.

Load a program on the AGV:

```
{
    "Program name": "MoveToAssemblyOperation",
    "State": 1
}
```

Execute the loaded program by sending a simple JSON object that forces state to 2, starting execution, e.g:

```
{
    "State": 2
}
```

For each PUT request you send, the asset will return status information. To request status information without using a PUT request, you can send a GET request, which will return a similar JSON response without any programs being executed. Example of return object format:

```
{
      "Battery": 42,
      "Program name": "MoveToAssemblyOperation",
      "State": 2,
      "TimeStamp": 12:34:56
}
```

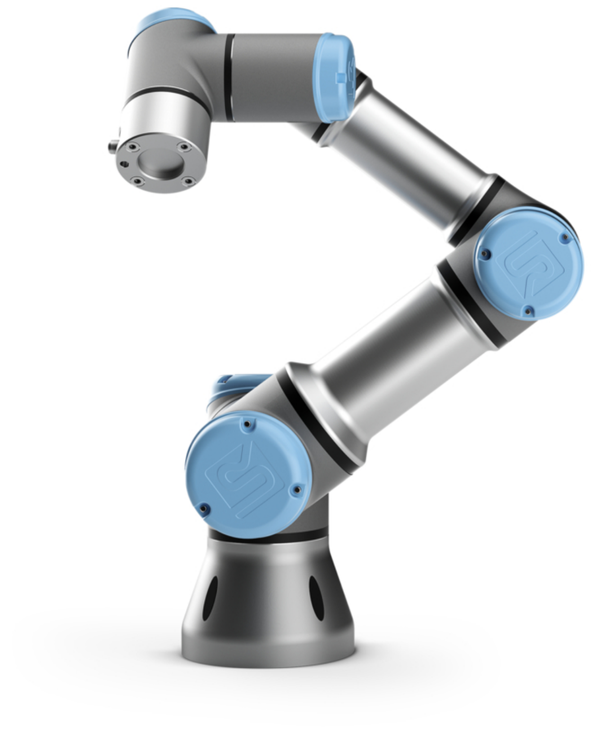
You can invoke the following methods by specifying the name in the PUT request:

Program name	**Description**
MoveToChargerOperation	Move the AGV to the charging station.
MoveToAssemblyOperation	Move the AGV to the assembly station.
MoveToStorageOperation	Move the AGV to the warehouse.
PutAssemblyOperation	Activate the robot arm to pick payload from AGV and place it at the assembly station.
PickAssemblyOperation	Activate the robot arm to pick payload at the assembly station and place it on the AGV.
PickWarehouseOperation	Activate the robot arm to pick payload from the warehouse outlet.
PutWarehouseOperation	Activate the robot arm to place an item at the warehouse inlet.

State information

State	**Description**
1	Idle
2	Executing
3	Charging

Assembly station



Asset communication

Assembly processes in the SDU I4.0 lab are handled in a variety of ways. There are automated processes where robots collaborate with other robots, collaborative processes where robots and humans work together to complete the task, and also processes that are handled by human hands only.

The processes involved in the current product line are strictly automatic, where robots work together to finish the assembly without manual input. We control these stations through a MQTT implementation, which is a scalable and lightweight solution that we can adapt to future requirement changes.

> *"**MQTT** (originally an initialism of Message Queueing Telemetry Transport) is a lightweight, publish-subscribe network protocol that transports messages between devices. The protocol usually runs over TCP/IP, however, any network protocol that provides ordered, lossless, bi-directional connections can support MQTT.[1] It is designed for connections with remote locations where resource constraints exist or the network bandwidth is limited. The protocol is an open OASIS standard and an ISO recommendation (ISO/IEC 20922)."*

- https://en.wikipedia.org/wiki/MQTT

Features and functionality

To call the methods available at the assembly station, you will need to implement a MQTT client that can talk to the broker which is hosted by the asset and can be accessed locally on port 1883. A typical URL would look something like **mqtt://localhost:1883/**.

The implementation of MQTT has three topics which you can subscribe to and publish messages to. Once subscribed to a topic, you should receive any updates that the asset publishes that topic.
Execution of the assembly process can be started by sending a process ID on the operation topic, whose value must be an integer. When using simulated assets you can specify any integer to start the assembly process.

You can publish and subscribe to the following topics:

Topic	**Description**

emulator/operation	Topic to execute programs.
	
	Input to start an operation must be JSON format, similar to:
	`{`
	`    "ProcessID": 12345`
	`}`
	
	To start an unhealthy assembly process you can publish “9999” to the topic. This can be used to verify correct error handling when subscribed to checkhealth.

emulator/status	The asset will broadcast frequent status messages on this topic in a JSON format, which is suitable for heartbeat/connection status etc.
	
	`Return example:`
	`{`
	`    "LastOperation": 1234,`

	`"CurrentOperation": 2345,` `"State": 1,` `"TimeStamp": 12:34:56` `}`
emulator/checkhealth	Result of the assembly process quality control. Result is published on the topic when the assembly process has finished, and will always be healthy for simulated processes unless forced.

State information

State	Description
0	Idle
1	Executing
2	Error

Deployment and usage

To run the services required for testing your software, ***you must have Docker installed*** on your machine to execute the associated images.
All the images will be publicly hosted on DockerHub, where you can pull them individually or use a docker-compose file to manage execution.
You can find an already defined docker-compose.yml file at https://github.com/ThMork/ST4-compose/blob/main/docker-compose.yml which will manage all the containers for you. Through your preferred terminal (cmd, powershell, git bash etc.) you can run the '**docker-compose up**' command in the directory of the file to initialize and run the images.

Note: If you have an m1 processor, you need to edit to docker-compose file to target the correct build. You can find the tag on DockerHub below. If you have no idea what this means, you likely won't need to change anything.

DockerHub images

- thmork/st4-warehouse
- thmork/st4-agv
- thmork/st4-assemblystation
- thmork/st4-mqtt

Content of docker-compose.yml

```
version: '3.8'

services:
  mqtt:
    image: thmork/st4-mqtt:latest
    ports:
      - 1883:1883
      - 9001:9001

  st4-agv:
    image: thmork/st4-agv:latest
    ports:
      - 8082:80

  st4-warehouse:
    image: thmork/st4-warehouse:latest
    ports:
      - 8081:80
```

```
st4-assemblystation:
  image: thmork/st4-assemblystation:latest
  environment:
    MQTT_TCP_CONNECTION_HOST: "mqtt"
    MQTT_TCP_CONNECTION_PORT: 1883
```

Development tools

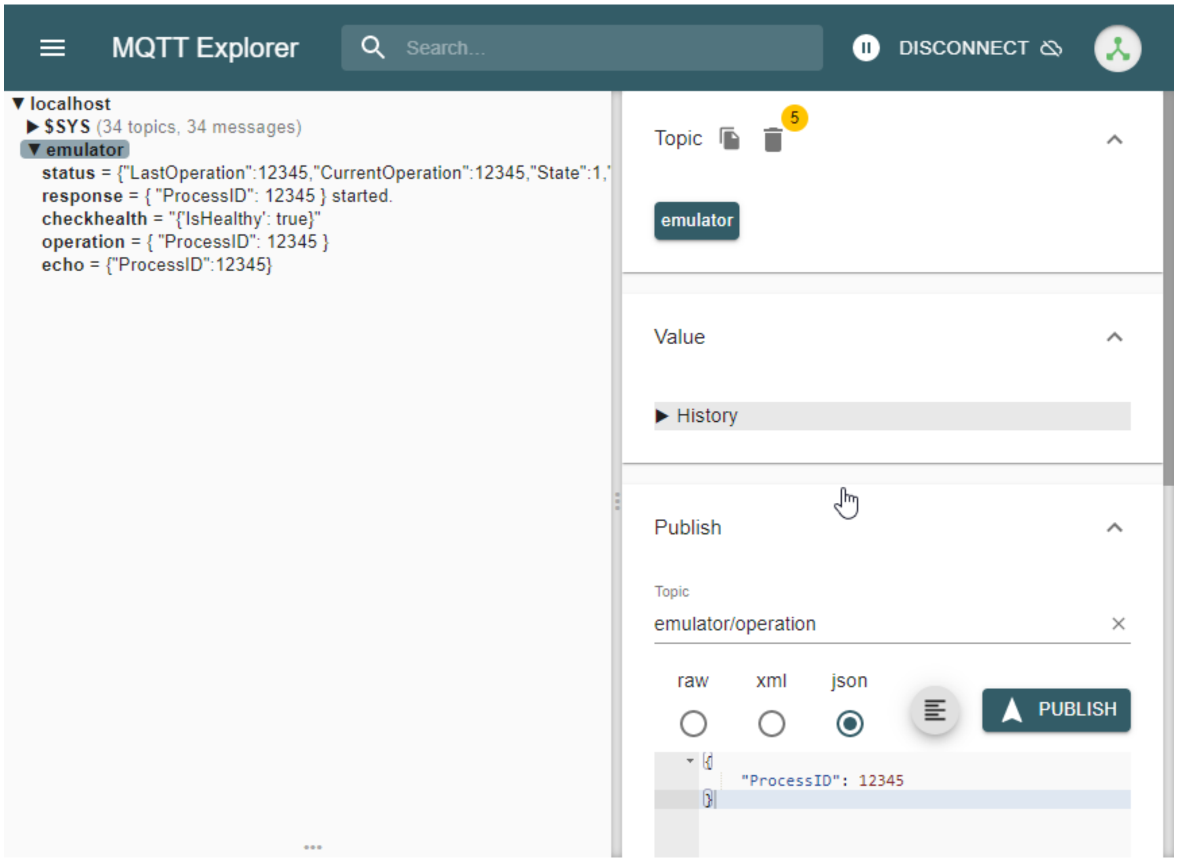
To help you verify the functionality of either containers or your code, and for exploring the APIs and functionality of these, you can use the following tools.

MQTT Explorer

Useful for exploring the MQTT broker, where you can see all different topics in use and publish messages on a given topic.

Download: http://mqtt-explorer.com/.

Example: Starting a process on assembly station from MQTT Explorer



```
▼ localhost
  ▶ $SYS (34 topics, 34 messages)
  ▼ emulator
    status = {"LastOperation":12345,"CurrentOperation":12345,"State":1,"
    response = { "ProcessID": 12345 } started.
    checkhealth = "{'IsHealthy': true}"
    operation = { "ProcessID": 12345 }
    echo = {"ProcessID":12345}
```

Topic: emulator/operation

json [✓]

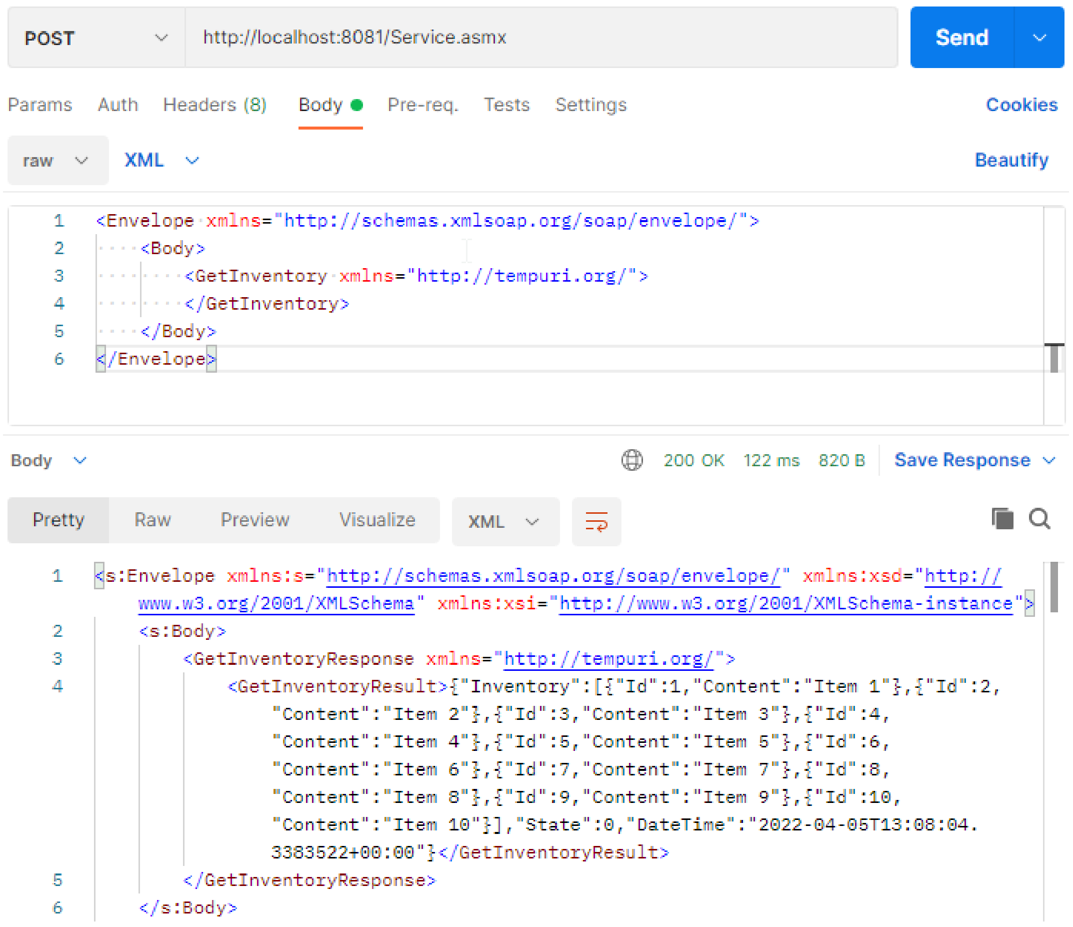
```
{
    "ProcessID": 12345
}
```

Postman

You can use Postman to send different types of http requests to an endpoint. In the ST4 project you can use it to verify the functionality of the AGV (get and put requests) and the Warehouse (construct raw xml objects to emulate implementation).

Download: https://www.postman.com/downloads/.

Example: Warehouse ‘GetInventory’ HTTP POST request



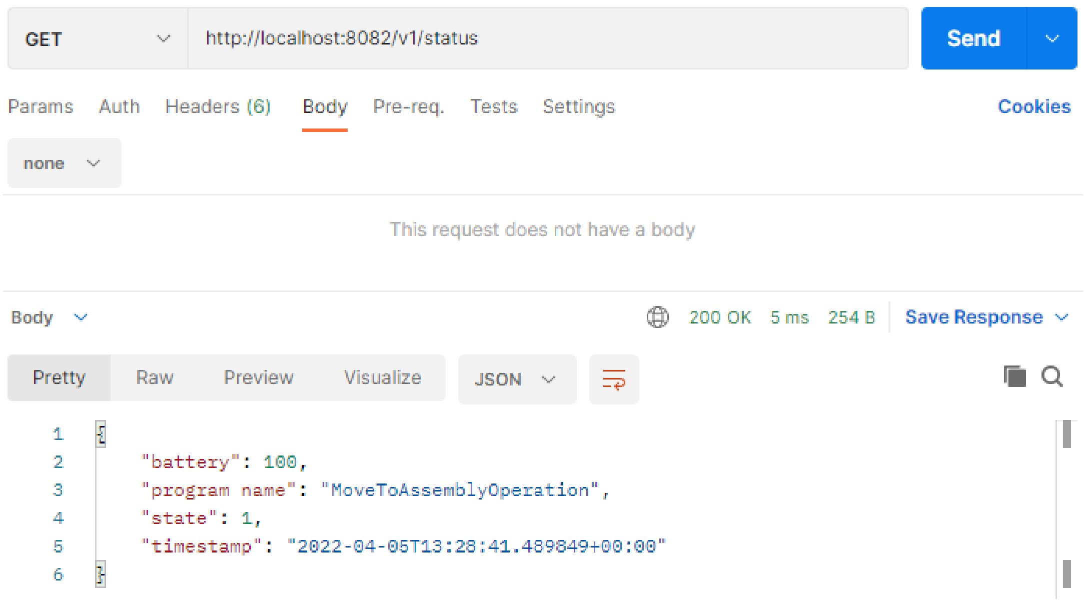
POST
http://localhost:8081/Service.asmx
Send
Params Auth Headers (8) Body Pre-req. Tests Settings
Cookies
raw XML
Beautify

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
    <Body>
        <GetInventory xmlns="http://tempuri.org/">
        </GetInventory>
    </Body>
</Envelope>
```

Body
200 OK 122 ms 820 B
Save Response
Pretty Raw Preview Visualize XML

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://
    www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <s:Body>
        <GetInventoryResponse xmlns="http://tempuri.org/">
            <GetInventoryResult>{"Inventory":[{"Id":1,"Content":"Item 1"},{"Id":2,
                "Content":"Item 2"},{"Id":3,"Content":"Item 3"},{"Id":4,
                "Content":"Item 4"},{"Id":5,"Content":"Item 5"},{"Id":6,
                "Content":"Item 6"},{"Id":7,"Content":"Item 7"},{"Id":8,
                "Content":"Item 8"},{"Id":9,"Content":"Item 9"},{"Id":10,
                "Content":"Item 10"}],"State":0,"DateTime":"2022-04-05T13:08:04.
                3383522+00:00"}</GetInventoryResult>
        </GetInventoryResponse>
    </s:Body>
```

Example: AGV status information by HTTP GET request



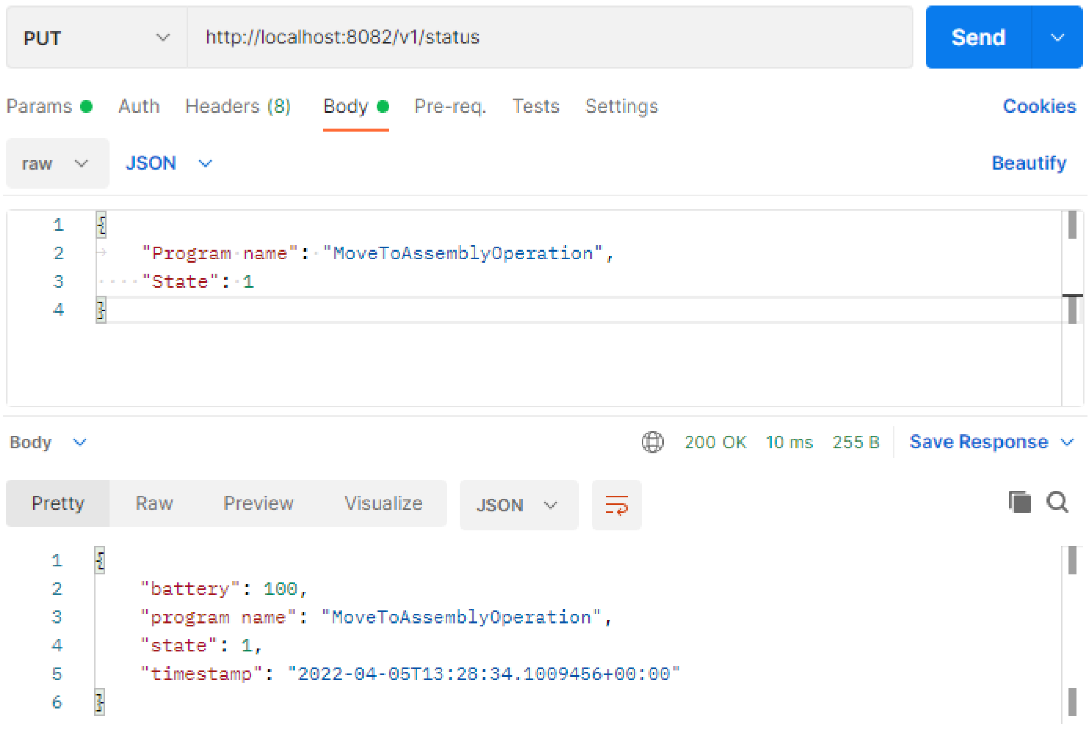
http://localhost:8082/v1/status

This request does not have a body

200 OK 5 ms 254 B

```
{
    "battery": 100,
    "program name": "MoveToAssemblyOperation",
    "state": 1,
    "timestamp": "2022-04-05T13:28:41.489849+00:00"
}
```

Example: AGV load ‘MovetoAssemblyOperation’ by HTTP PUT request



http://localhost:8082/v1/status

```
{
    "Program name": "MoveToAssemblyOperation",
    "State": 1
}
```

200 OK 10 ms 255 B

```
{
    "battery": 100,
    "program name": "MoveToAssemblyOperation",
    "state": 1,
    "timestamp": "2022-04-05T13:28:34.1009456+00:00"
}
```