REST APIs

Carmine Tommaso Recchiuto

Cloud Robotics

Cloud services, in recent years, have strongly impacted also the world of robotics: robots and automation systems can rely on data or code from a network to support their operation, i.e., **not all sensing, computation, and memory is integrated into a standalone system**

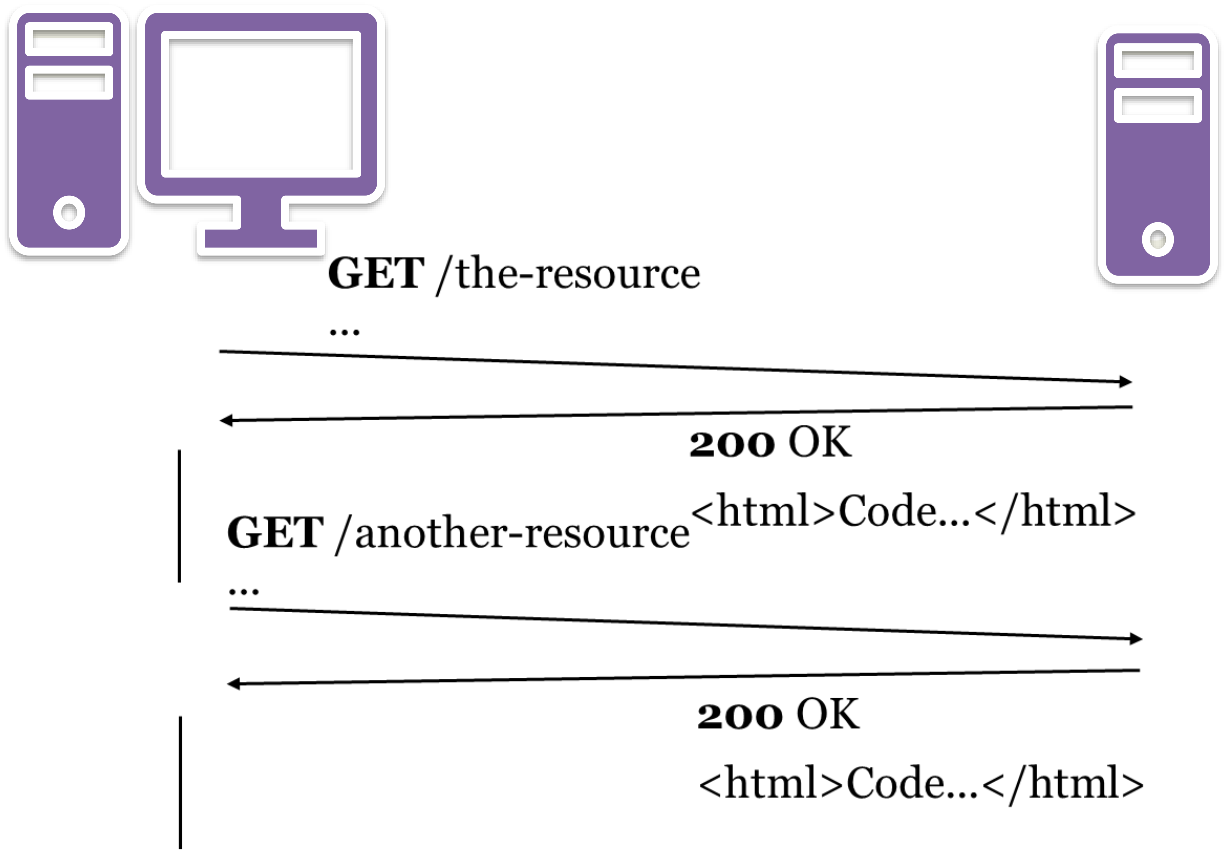
There are mainly 4 potential benefits of using this kind of infrastructure, which have been reflected in existing applications in robotics:

- ✓ **Big Data:** access to libraries of images, maps, trajectories, data
- ✓ **Cloud Computing:** access to parallel grid computing for statistical analysis, learning, motion planning
- ✓ **Collective Learning:** robots sharing trajectories, control policies and outcomes
- ✓ **Human Computation:** crowdsourcing to analyze images and video, classification, learning, error recovery

Cloud Robotics

In this context, finding a simple but easy way for letting the different nodes of the architecture communicate with each other is essential.

Traditional Web Applications



Cloud Robotics

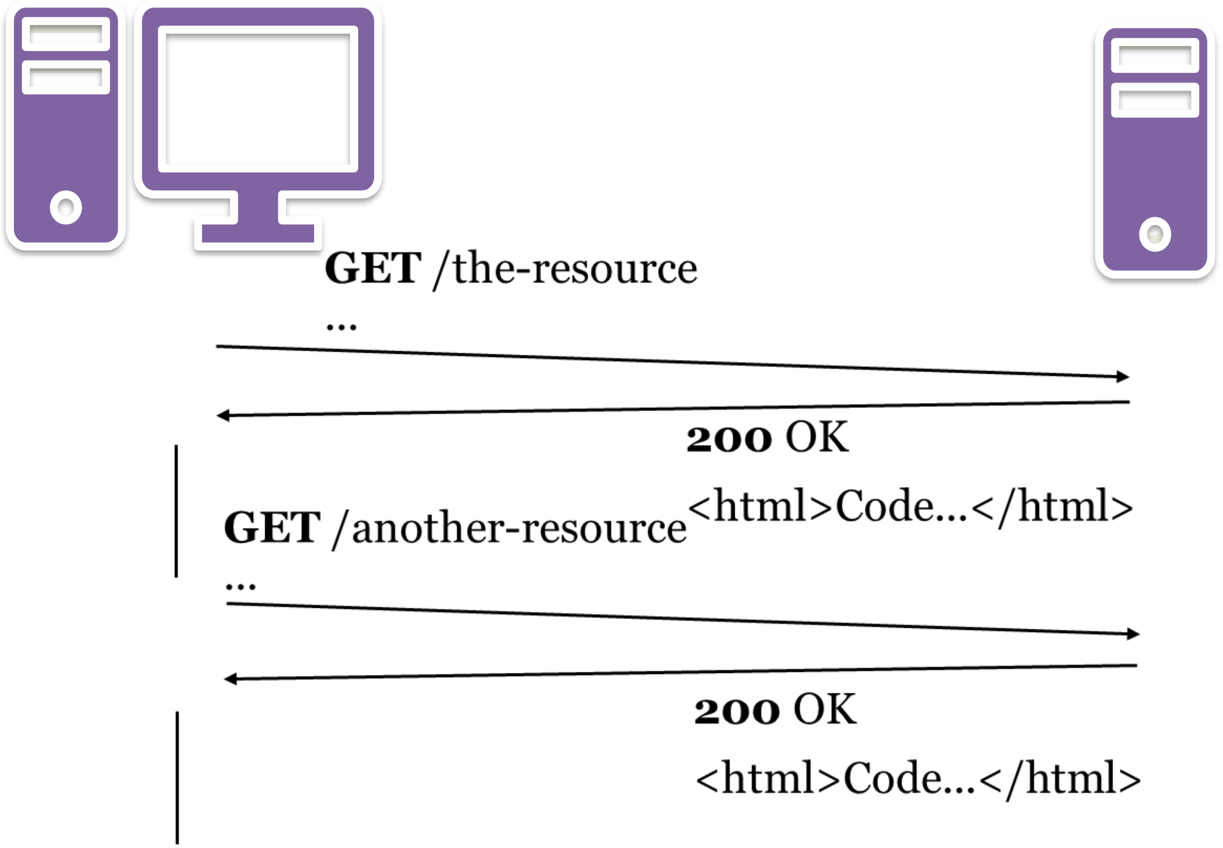
Traditional Web Applications

Limits:

- The interface should be built on HTML & HTTP, which limits the integration with popular frameworks for robotics (ROS*)

- Data is usually sent in multiple responses.

As we will see, Rosbridge_server provides a WebSocket transport layer for continuous communication!



Application Programming Interfaces

A different way to let two applications to communicate with each other, without relying on HTML.

It acts as an intermediary layer that processes data transfers between systems.

To work with different systems, there should be a set of definitions and protocols that have to be used:

Example:

- Weather service which offers an API to retrieve weather data
- The weather app on the phone that uses the API to show daily weather update on the phone

An API making use of the HTTP is called *Web API*

How do API works?

API architecture is usually explained in terms of client and server. The application sending the request is called the client, and the application sending the response is called the server. So in the weather example, the bureau’s weather database is the server, and the mobile app is the client.

There are four different ways that Web APIs can work depending on when and why they were created.

1 - SOAP APIs
These APIs use Simple Object Access Protocol. Client and server exchange messages using XML. This is a less flexible API that was more popular in the past.

2- RPC APIs
These APIs are called Remote Procedure Calls. The client completes a function (or procedure) on the server, and the server sends the output back to the client.

How do API works?

3 - Websocket APIs
Websocket APIs is another modern web API development that uses JSON objects to pass data. A WebSocket API supports two-way communication between client apps and the server. The server can send callback messages to connected clients, making it more efficient than REST API.

4 - REST APIs
These are the most popular and flexible APIs found on the web today. The client sends requests to the server as data. The server uses this client input to start internal functions and returns output data back to the client. Let’s look at REST APIs in more detail below, but after some technical considerations.

XML

eXtensible Markup Language (XML)

A set of rules for encoding documents electronically.

De-facto standard (W3C Recommendation).

It is usually not very human-readable, with a well-defined scheme (vocabulary, content models, datatypes) and the usage of URIs

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Person>
  <firstName>John</firstName>
  <lastName>Smith</lastName>
  <age>25</age>
  <address>
    <streetAddress>21 2nd Street</streetAddress>
    <city>New York</city>
    <state>NY</state>
    <postalCode>10021</postalCode>
  </address>
  <phoneNumber type="home">212 555-1234</phoneNumber>
  <phoneNumber type="fax">646 555-4567</phoneNumber>
  <newSubscription>false</newSubscription>
  <companyName />
</Person>
```

JSON

JavaScript Object Notation (JSON)
A lightweight computer data interchange format.

Represents a simple alternative to XML, being a text-based, human-readable format for representing simple data structures and associative arrays (called objects).

Used by a growing number of services (Web APIs)

No namespaces, attributes etc.
No schema language (for description, verification)

```
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021"
  },
  "phoneNumbers": [
    { "type": "home", "number": "212 555-1234" },
    { "type": "fax", "number": "646 555-4567" }
  ],
  "newSubscription": false,
  "companyName": null
}
```

REST APIs

- **Representational State Transfer (REST)**
  - A style of software architecture for distributed hypermedia systems such as the World Wide Web.
- **REST is basically client/server architectural style**
  - Requests and responses are built around the transfer of "representations" of "resources".
  - Resources are sources of specific information, referenced with a global identifier (e.g., a URI in HTTP).
- **Components of the network (user agents and origin servers) communicate via a standardized interface (e.g., HTTP)**
  - exchange representations of these resources (the actual documents conveying the information).

REST APIs

- **REST defines a set of architectural rules. The main three rules are:**

1 - Client-server

- *Separation of concerns*
  - Clients are separated from servers by a uniform interface.
- *Networking*
  - Clients are not concerned with data storage, which remains internal to each server, so that the portability of client code is improved. Servers are not concerned with the user interface or user state, so that servers can be simpler and more scalable.
- *Independent evolution*
  - Servers and clients may also be replaced and developed independently, as long as the interface is not altered.

REST APIs

- **REST defines a set of architectural rules. The main three rules are:**

2 - Stateless communication

- *Scalability, reliability*
  - No client context being stored on the server between requests. Each request from any client contains all of the information necessary to service the request.
- *Resources are conversationally stateless*
  - Any conversational state is held in the client.

3 - Uniform Interface

- Example: Create, Retrieve, Update, Delete

HTTP as Uniform Interface

- **Use URIs to identify resources** (e.g. http://130.251.13.192:3000/users)
- **Use HTTP methods to specify operation**:
  - Create: POST (or PUT)
  - Retrieve: GET
  - Update: PUT (or PATCH)
  - Delete: DELETE
- **Use HTTP status code** to indicate success/failure

e.g., GET http://130.251.13.192:3000/users

e.g., HTTP/1.1 200 OK

GET Method

- **Description:** The GET method is used to retrieve information from the server. It should only retrieve data and have no other effect on the data.
- **Example:**
    - **Endpoint: /api/users**
    - **Request: GET /api/users**
    - **Response:**

```
{
  "users": [
    {"id": 1, "name": "John Doe"},
    {"id": 2, "name": "Jane Smith"}
  ]
}
```

PUT Method

- **Description:** The PUT method is used to update a resource on the server. It replaces the existing resource or creates a new one if it doesn't exist.
- **Example:**
  - **Endpoint: /api/users/1**
  - **Request: PUT /api/users/1**

```
{
"name": "Updated Name"
}
```

  - **Response:**

```
{
  "id": 1,
  "name": "Updated Name"
}
```

POST Method

Description: The POST method is used to submit data to be processed to a specified resource. It is often used to create new resources.

Example:

- **Endpoint: /api/users**
- **Request: POST /api/users**

```
{
  "name": "New User"
}
```

- **Response:**

```
{
  "id": 3,
  "name": "New User"
}
```

DELETE Method

Description: The DELETE method is used to request the removal of a resource on the server.

Example:

- **Endpoint: /api/users/2**
- **Request: DELETE /api/users/2**
- **Response:**

```
{
  "message": "User with ID 2 deleted successfully."
}
```

Practical Implementation: FLASK

Flask is a lightweight web framework for Python that makes it easy to create REST APIs.

Why Flask?

- Flask is easy to learn and quick to set up, making it an excellent choice for building small to medium-sized APIs.
- It comes with built-in development server and debugger.

Installation:

pip (or pip3) install Flask.

Practical Implementation: FLASK

A minimal Flask application:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
   return 'Hello, World!'

if __name__ == '__main__':
   app.run(debug=True)
```

- Flask(__name__): Creates a Flask web server.
- @app.route('/'): Decorator* that associates the function with the specified route.
- def hello_world(): The function to be executed when the specified route is accessed.
- app.run(host='127.0.0.1', port=3000, debug=True): Runs the Flask server.

*a wrapper which embeds a function into another function

Practical Implementation: FLASK

Routes may be defined for different HTTP methods (GET, POST, PUT, DELETE):

```
@app.route('/api/users', methods=['GET'])
def get_users():
    # Code to get and return a list of users

@app.route('/api/users', methods=['POST'])
def create_user():
    # Code to create a new user

@app.route('/api/users/<int:user_id>', methods=['GET', 'PUT',
'DELETE'])
def manage_user(user_id):
    # Code to get, update, or delete a specific user
```

<variable_type:variable_name> can be used to capture values from the URL.

Practical Implementation: FLASK

Within routes, I can handle JSON data from the request

```
from flask import request

@app.route('/api/users', methods=['POST'])
def create_user():
    user_data = request.json
    # Code to create a new user using user_data
```

While also return values are expressed as JSON data

```
from flask import jsonify

@app.route('/api/users', methods=['GET'])
def get_users():
    users = get_users_from_database()
    return jsonify({'users': users})
```

FLASK and Robotics

- ✓ **Ease of Integration:**
  Flask's lightweight and modular structure simplifies integration with existing robotic systems.

- ✓ **Remote Robot Control:**
  Using Flask allows for remote control of robots by exposing APIs over the network.

- ✓ **Sensor Data Retrieval:**
  Flask can be used to retrieve sensor data from robots, providing a standardized interface for data exchange. Depending on the type of sensor, however, REST APIs may not be the best choice (websockets may be more suitable for the scope).

- ✓ **Scalability:**
  Flask's scalability makes it suitable for applications ranging from small robotic prototypes to large-scale robotic systems.

FLASK and Robotics

- **REST APIs** are a universal standard. They enable web, mobile, cloud, or legacy software applications to easily communicate with ROS.

- ROS itself uses many ports (nodes, topics, etc.), complicating remote access.
  - **REST APIs** typically use HTTP/HTTPS on a single port, making it easier to:
    - access from WAN
    - cross firewalls
    - deploy in cloud environments

On the other hand, ror very frequent or real-time operations (e.g., high-frequency sensor streaming), REST is not optimal. In these cases, WebSocket or rosbridge/ROS2 DDS for high-frequency streams is more indicated

https://github.com/CarmineD8/rest_api_ros/

FLASK and Robotics

```
import threading
from flask import Flask, request, jsonify

import rclpy
from rclpy.node import Node
from rclpy.action import ActionClient

from nav2_msgs.action import NavigateToPose
from geometry_msgs.msg import PoseStamped

# -------------------------
# ROS2 NODE
# -------------------------
class Nav2API(Node):
    def __init__(self):
        super().__init__("rest_api_gateway")
        self.action_client = ActionClient(self, NavigateToPose, 'navigate_to_pose')


app = Flask(__name__)
rclpy.init()
node = Nav2API()
```

Used so ROS2 can run in a *separate thread* while Flask runs the web server.
ROS2 needs a running "spin" loop, and Flask needs its own event loop → two separate threads.

ROS2 node class

Creates an **Action Client** that connects to the Nav2 action server

FLASK and Robotics

```
@app.route("/go_to", methods=["POST"])
def go_to():

    # retrieving parameters
    x = float(request.args.get("x", 0.0))
    y = float(request.args.get("y", 0.0))
    yaw = float(request.args.get("yaw", 0.0))

    if not node.action_client.wait_for_server(timeout_sec=2.0):
        return jsonify({"error": "NavigateToPose Action Server not available"})

    goal_msg = NavigateToPose.Goal()
    pose = PoseStamped()
    pose.header.frame_id = "map"
    pose.pose.position.x = x
    pose.pose.position.y = y
    pose.pose.orientation.w = 1.0  # Yaw semplificato

    goal_msg.pose = pose
```

Route for integrating with the navigation server

```
# Send goal
    future_goal = node.action_client.send_goal_async(goal_msg)
    rclpy.spin_until_future_complete(node, future_goal)

    goal_handle = future_goal.result()
    if not goal_handle.accepted:
        return jsonify({"status": "Goal rejected"})

    # Aspetta il risultato
    future_result = goal_handle.get_result_async()
    rclpy.spin_until_future_complete(node, future_result)

    result = future_result.result()

    return jsonify({"status": "Goal finished", "result": str(result)})
```

FLASK and Robotics

```
def ros_spin():
    rclpy.spin(node)


if __name__ == "__main__":

    # Thread for ROS2
    ros_thread = threading.Thread(target=ros_spin, daemon=True)
    ros_thread.start()

    ip = "0.0.0.0"
    port = 8000

    print("\n========================================")
    print("  ROS2 + REST API NAVIGATION GATEWAY  ")
    print("========================================")
    print(f"API on: http://{ip}:{port}")
    print("========================================\n")

    # Starting Flask
    app.run(host=ip, port=port)
```

Thread ROS

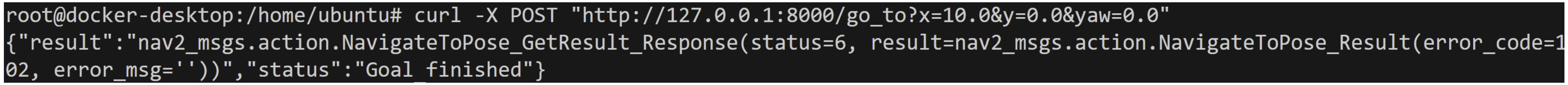
Starting the ROS thread

Starting the Flask application

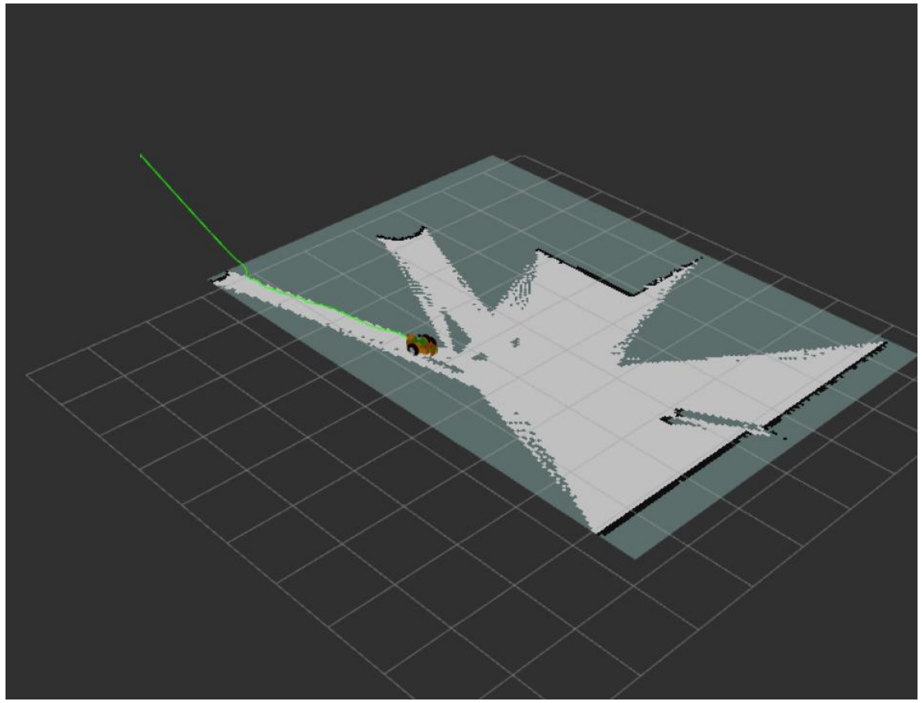
FLASK and Robotics

We can test the API with curl:

curl -X POST "http://127.0.0.1:8000/go_to?x=1.0&y=2.0&yaw=0.0"



```
root@docker-desktop:/home/ubuntu# curl -X POST "http://127.0.0.1:8000/go_to?x=10.0&y=0.0&yaw=0.0"
{"result":"nav2_msgs.action.NavigateToPose_GetResult_Response(status=6, result=nav2_msgs.action.NavigateToPose_Result(error_code=1
02, error_msg=''))","status":"Goal_finished"}
```



FLASK and Robotics

While it’s possible to send robot’s control with curl, REST APIs give the possibility to use more user-friendly interfaces (sending_goals.html):

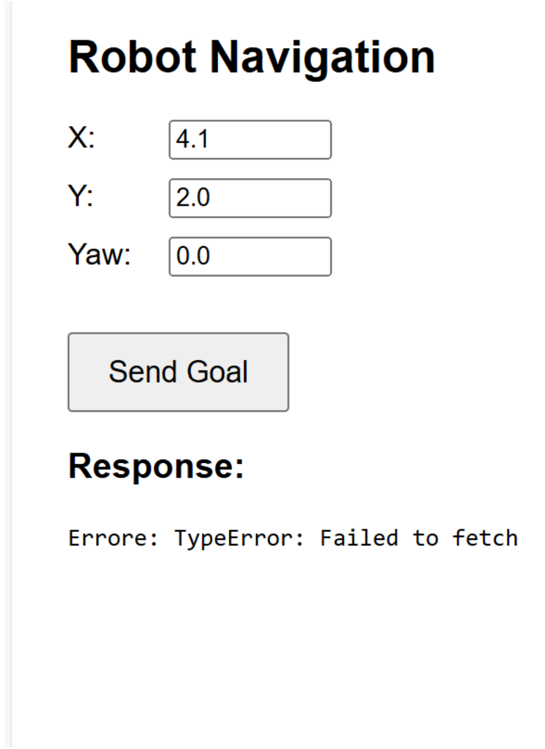
- HTML

```
<h2>Robot Navigation</h2>

<label>X:</label>
<input id="x" type="number" step="0.1" value="1.0"><br>

<label>Y:</label>
<input id="y" type="number" step="0.1" value="2.0"><br>

<label>Yaw:</label>
<input id="yaw" type="number" step="0.1" value="0.0"><br>

<button onclick="sendGoal()">Send Goal</button>

<h3>Response:</h3>
<pre id="response">No requests</pre>

<script>
function sendGoal() {
  const x   = document.getElementById("x").value;
  const y   = document.getElementById("y").value;
  const yaw = document.getElementById("yaw").value;

const url = `http://127.0.0.1:8000/go_to?x=${x}&y=${y}&yaw=${yaw}`;

  // Send POST request
  fetch(url, { method: "POST" })
    .then(res => res.json())
    .then(data => {
        document.getElementById("response").innerText =
JSON.stringify(data, null, 2);
    })
    .catch(err => {
        document.getElementById("response").innerText = "Errore: " + err;
    });
}
</script>

</body>
</html>
```

FLASK and Robotics

The server's IP could be used for driving the robot even remotely.

For example, even if I don't have ROS2 on my host system (windows) and therefore I could not directly teleoperate the robot from there, I could use the web browser for sending goals

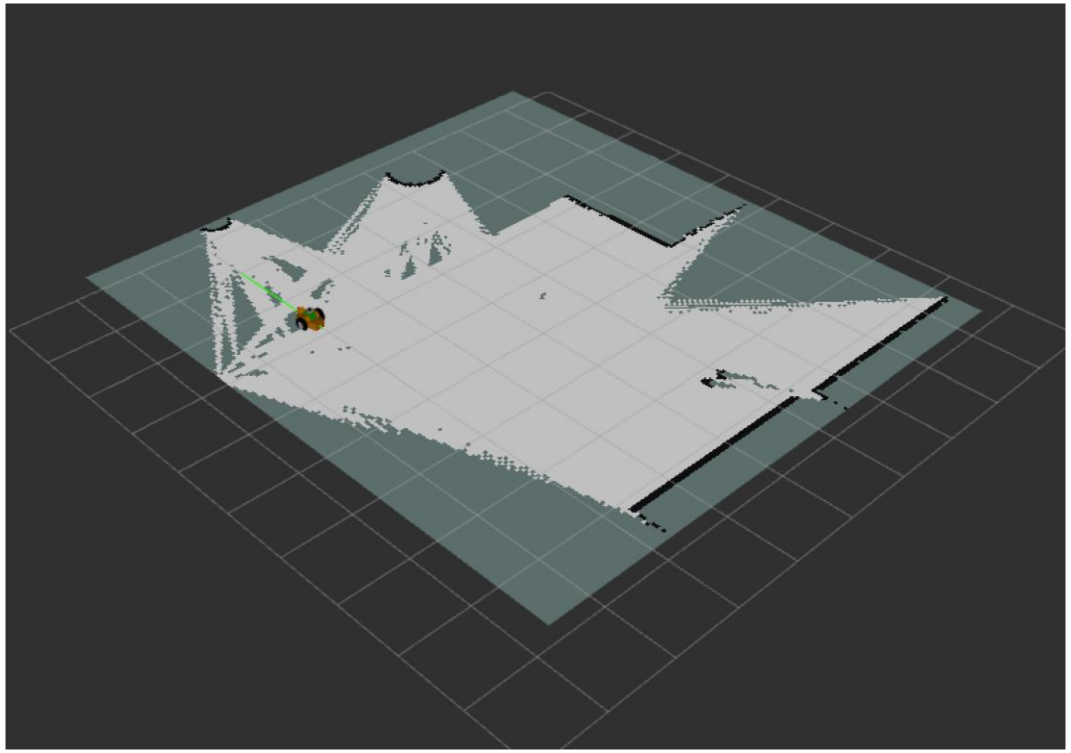


Robot Navigation

X:	4.1
Y:	2.0
Yaw:	0.0

Response:

```
Errore: TypeError: Failed to fetch
```



Websockets and Robotics

What about received data from the robot?
As mentioned before, in this case, Flask is not the optimal choice, and it may be better to use Websocket

A **WebSocket** is a communication protocol that creates a **persistent, two-way connection** between a client (like a web browser) and a server.

Unlike HTTP, which is **request → response**, WebSocket allows:

- the client to send data to the server **at any time**
- the server to send data to the client **at any time**

This makes WebSockets better suited for continuous communication!

Websockets and Robotics

Ros(2) has already an integrated package named “rosbridge_suite”, that forwards data from the ROS environment to any external application.

We can install it with apt-get install ros-jazzy-rosbridge-suite

By default, the system forwards data on the port 9090 (ros2 launch rosbridge_server rosbridge_websocket_launch.xml)

rosbridge_websocket_launch.xml (please notice that you may need to change the parameter delay_between_messages to 0.0)

Websockets and Robotics

On the other side, an HTML + JavaScript program can **connect to ROS** through **rosbridge (WebSocket)**, subscribe to the **/odom** topic, and extract all relevant information (odom_viewer.html)

```
var ros = new ROSLIB.Ros({
   url : 'ws://127.0.0.1:9090'
});
```

$\longrightarrow$ This attempts to open a WebSocket connection to rosbridge running locally.

```
ros.on('connection', ...)
ros.on('error', ...)
ros.on('close', ...)
```

$\longrightarrow$ Event handlers: on-screen status message depending on what happens.

```
var odomListener = new ROSLIB.Topic({
   ros : ros,
   name : '/odom',
   messageType : 'nav_msgs/msg/Odometry'
});
```

$\longrightarrow$ Subscriber to odom