Robot Radar

Software Requirements Specification (SRS) Template v1.0

[Instructions Specific to the course SRS including formatting and any notable notation.]

This document is an annotated outline intended for specifying software requirements and is adapted from IEEE 29148-2018.

Version 0.0.0.1
Prepared By: Group C
Prepared For CSC-380 SUNY Oswego
[Date]

Table of Contents

Change Log

Name	Date	Reason For Changes	Version
Victor Lockwood	8/31/2022	Reworded the scope so as not to be directly pulled from the project description	1.0.1
Ethan Uliano	9/8/22	Added Scenario, user stories, and personas	1.0.2
Adam Durand	9/8/22	Added Scenario, user stories, and personas	1.0.2
Victor Lockwood	9/8/22	Added Scenario, user stories, and personas	1.0.2
Jabel Alcantara	9/8/22	Added Scenario, user stories, and personas	1.0.2

Victor Lockwood	9/23/2022	Updated UML diagram links and references	1.0.3

1. Introduction

1.1 Purpose

For Bastian who is using this for his research, the Robot Radar is an obstacle tracking system that provides a front-end application to extend the GoPiGo platform and offer features that no other proprietary GoPiGo2 systems include, such as obstacle detection capabilities relative and absolute to the robot's position, as well as displaying those obstacles on a map.

1.2 Scope

This document will not contain any actual code implementation but will keep an overview of the technologies used and what is to be expected of the project. Using a GoPiGo2 setup with a radar sensor, we will create an application to detect obstacles. We'll use Dijkstra's algorithm to generate the shortest path between vertices in a map, display a 360 panoramic image and finally, offer an autonomous mode to move the robot. This project will not be the robot itself, it will be the software.

1.3 Product Overview

1.3.1 Product Perspective

1.3.1.1 System Interfaces

Raspberry Pi 3 - Computer running the detection software
Radar Sensor - The sensor that will detect obstacles
The GoPiGo2 board
Camera

1.3.1.2 User Interfaces

Web page

1.3.1.3 Hardware Interfaces

The web UI will take input from a mouse and keyboard and will support computers that can run the latest versions of Chrome, Firefox and Safari.

The robot software will be meant to run on a Raspberry Pi 3 B+ with the latest version of Raspbian as well as a radar sensor, line sensor, GoPiGo2 board, and the Raspberry Pi camera module.

1.3.1.4 Software Interfaces

Raspbian OS - Version 5.15
Python - Version 3.11
Flask - Version 2.2.2
PostgreSQL - Version 15

1.3.1.5 Communication Interfaces

We will have a server that uses a REST API. The web interface and the robots will communicate back and forth with this API.

The web interface will take input from a keyboard and a mouse.

The robots will be connected to the Colosseum WiFi network.

1.3.1.6 Memory Constraints

1GB RAM
16 GB Storage
1TB Memory of server (Moxie)

1.3.1.7 Site Adaptation

ECE lab

- Any location where a 5m x 5m site exists with obstacles limited to the size of a soda can

1.3.1.8 Interfaces with Services

APIs and server-side code will be hosted on the Moxie server.

1.3.2 Product Functions

- Control the GoPiGo2 (move) robots and read sensor data.
- Display obstacles, (i.e. showing them on a screen).

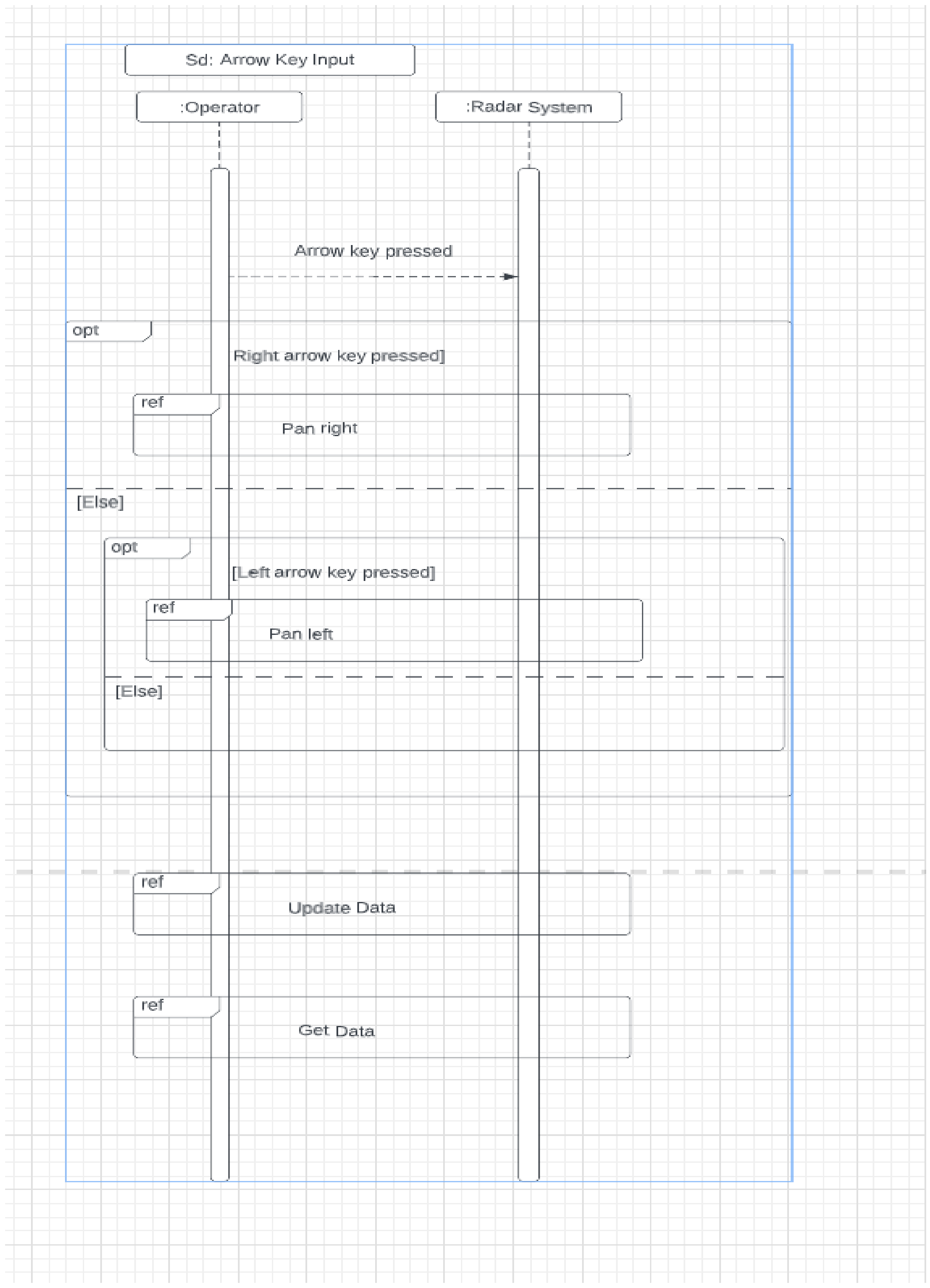
- Robot with camera will take a 360 panorama image
- Account of robot and obstacle movement, absolute and relative to the “own” robot’s position.
- Differentiating separate obstacles
- Create a map based on the physical 5m x 5m map, which is generated by location and obstacle data on a web app interface via a JSON response from a REST API that communicates between a database and the robots in the field.

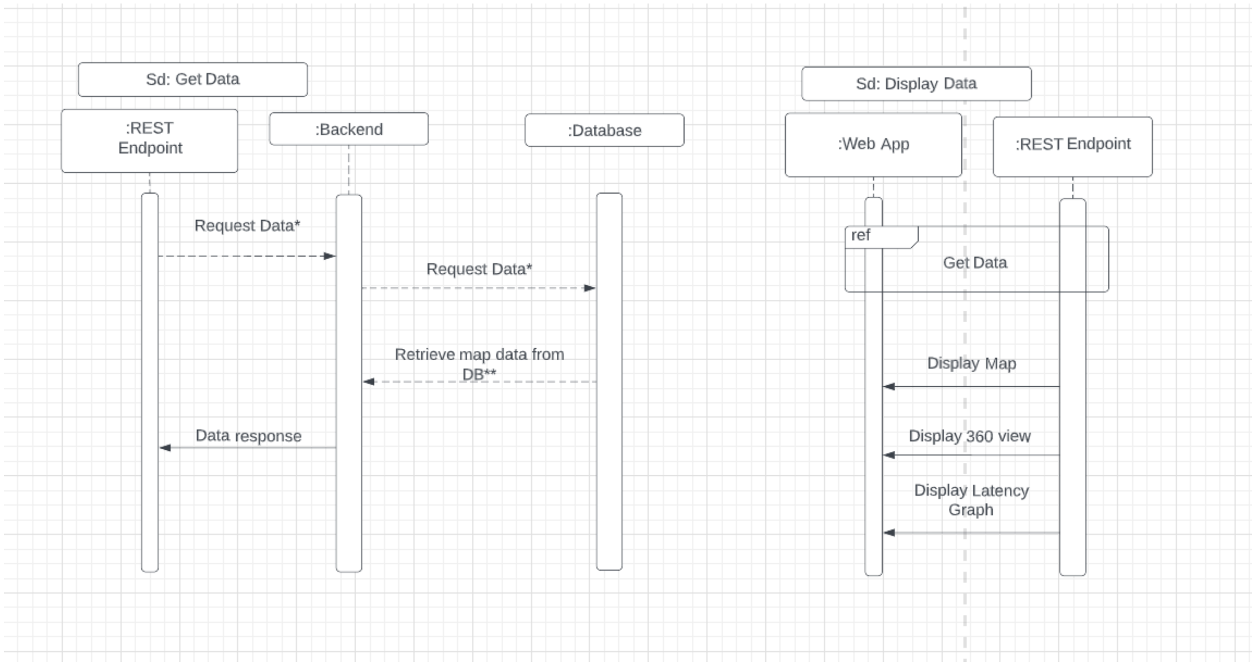
1.3.3 Features

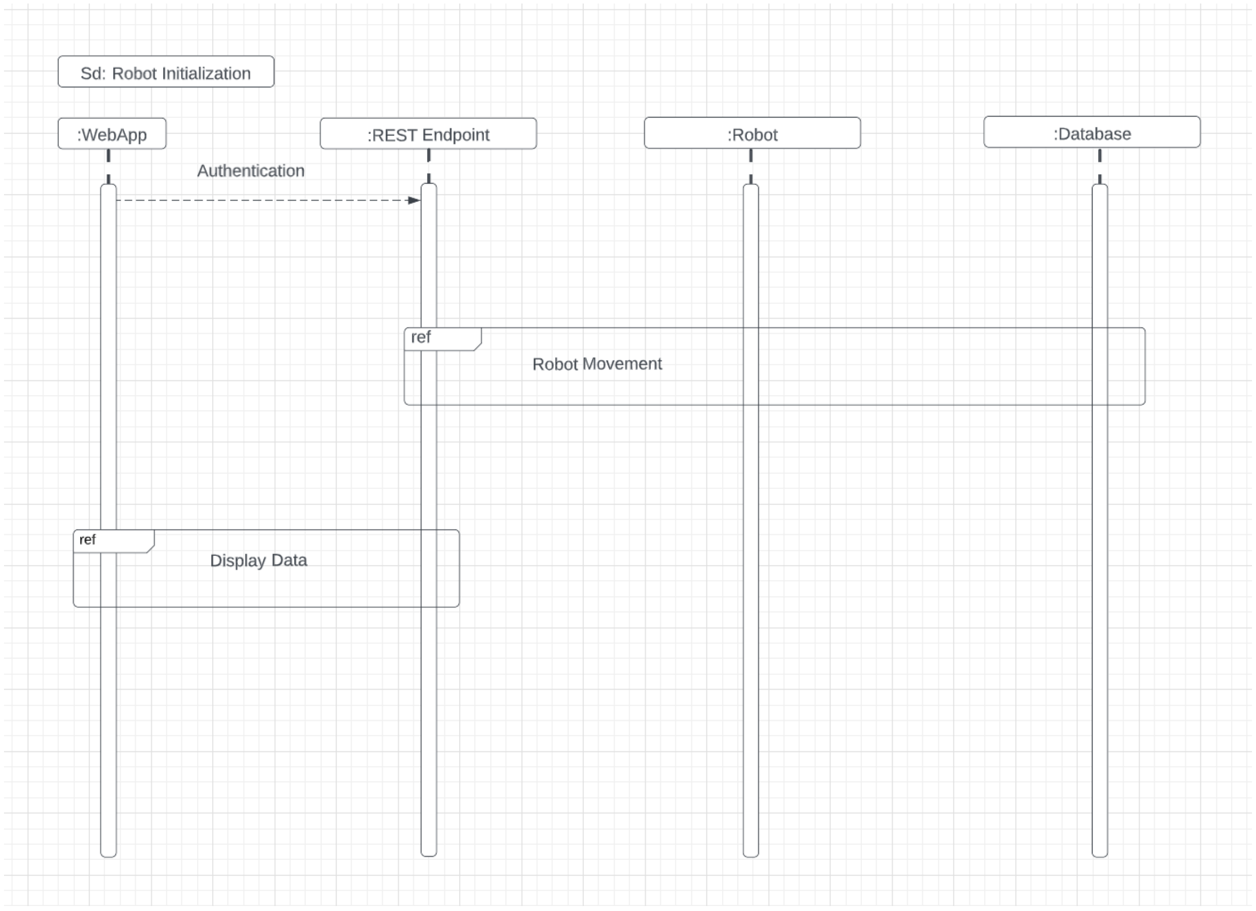
- Control the movement of the robot
- Visualize obstacles in a given field on a display
- Take panoramic images of surrounding area
- Communicate obstacles between robots
- Self-position tracking
- Position tracking of another robot
- User authentication
- Autonomous mode
- Web application
- Have multiple browser windows open to control multiple robots (only the active window should take user input)
- Calculate position
- Autonomous mode

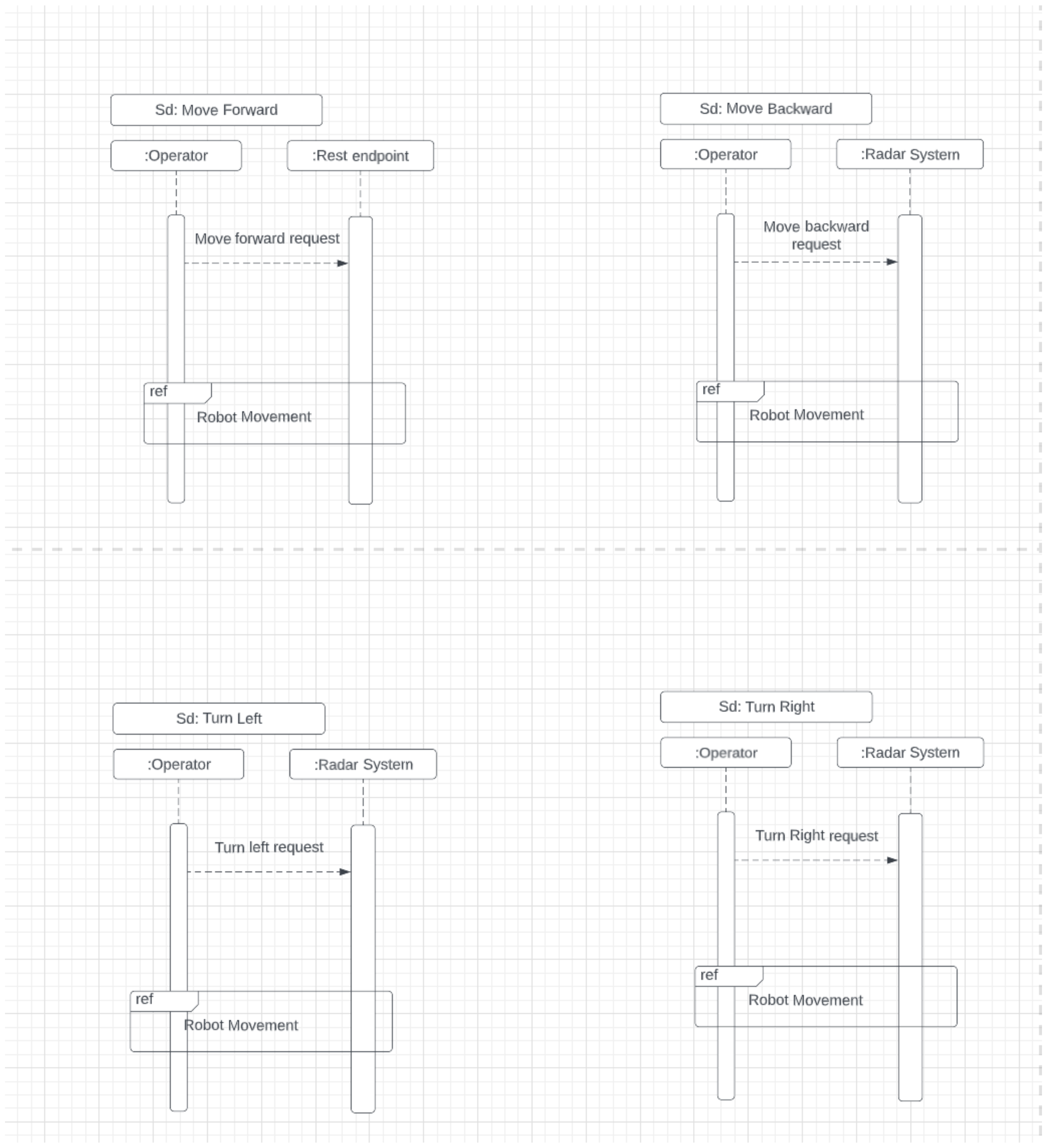
UML Models
- Master Page
- Activity Diagrams

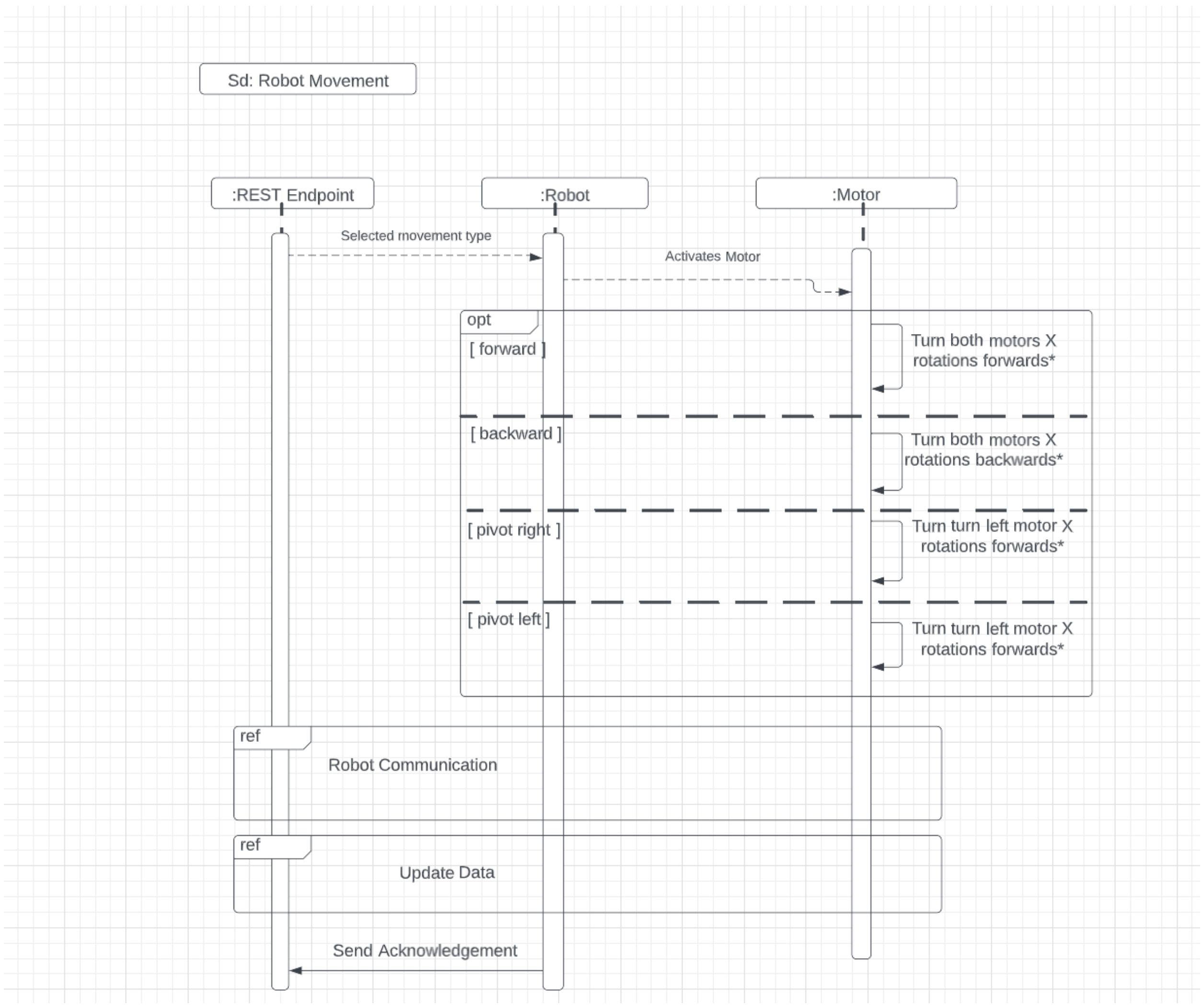
Activity Diagrams

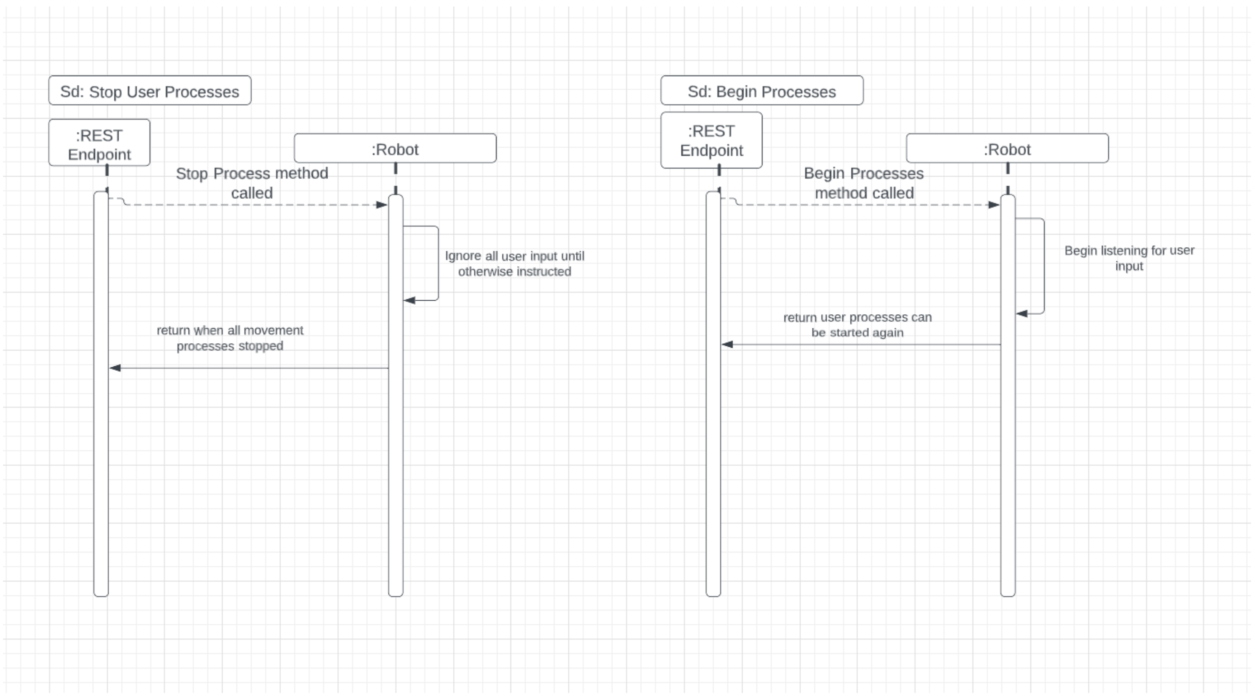


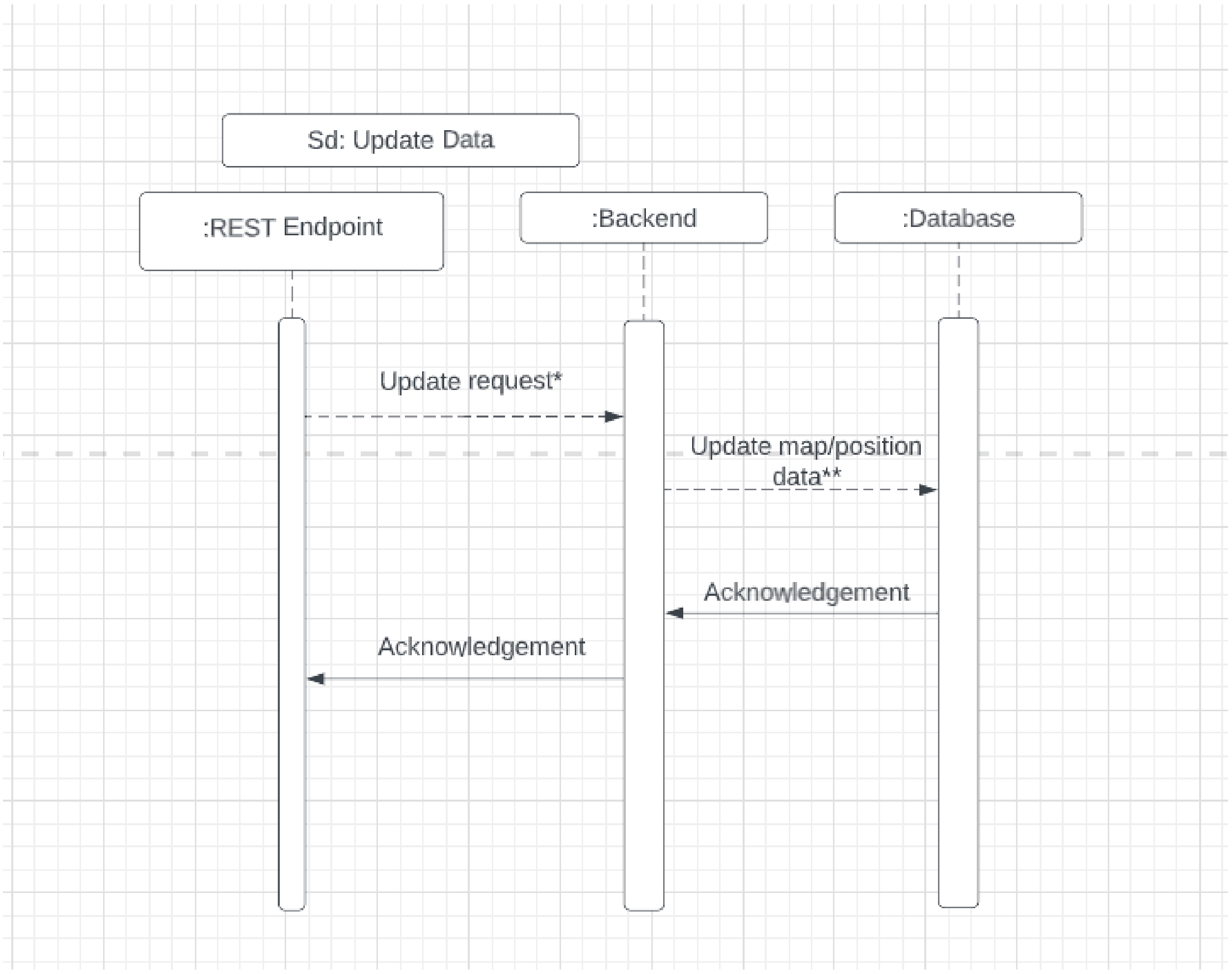

Sd: Get Data
:REST Endpoint
:Backend
:Database
Request Data*
Request Data*
Retrieve map data from DB**
Data response
Sd: Display Data
:Web App
:REST Endpoint
ref
Get Data
Display Map
Display 360 view
Display Latency Graph


Sd: Robot Initialization
:WebApp
:REST Endpoint
:Robot
:Database
Authentication
ref
Robot Movement
ref
Display Data

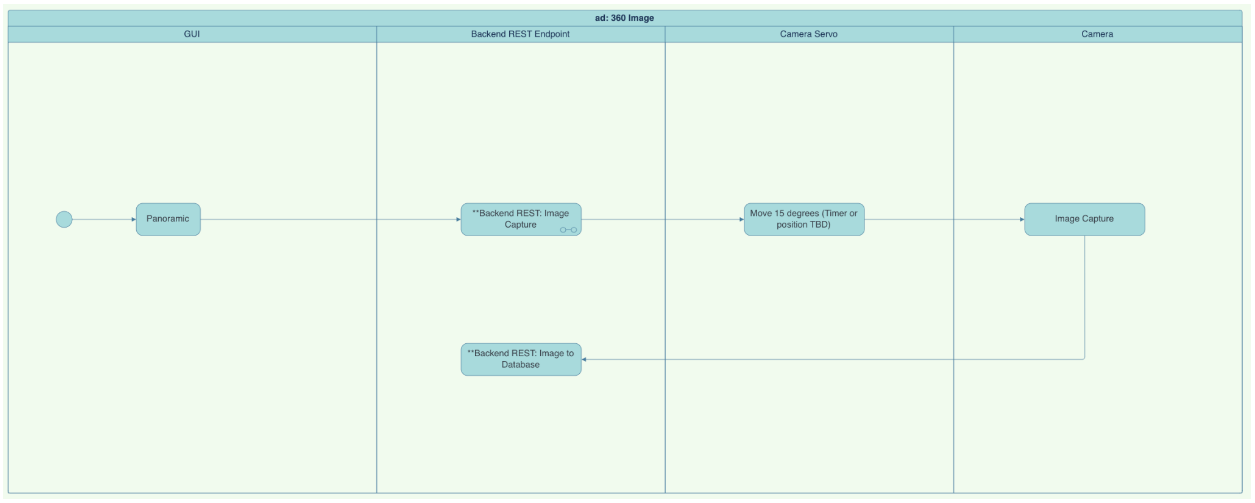

Sd: Move Forward
:Operator
:Rest endpoint
Move forward request
ref
Robot Movement
Sd: Move Backward
:Operator
:Radar System
Move backward request
ref
Robot Movement
Sd: Turn Left
:Operator
:Radar System
Turn left request
ref
Robot Movement
Sd: Turn Right
:Operator
:Radar System
Turn Right request
ref
Robot Movement

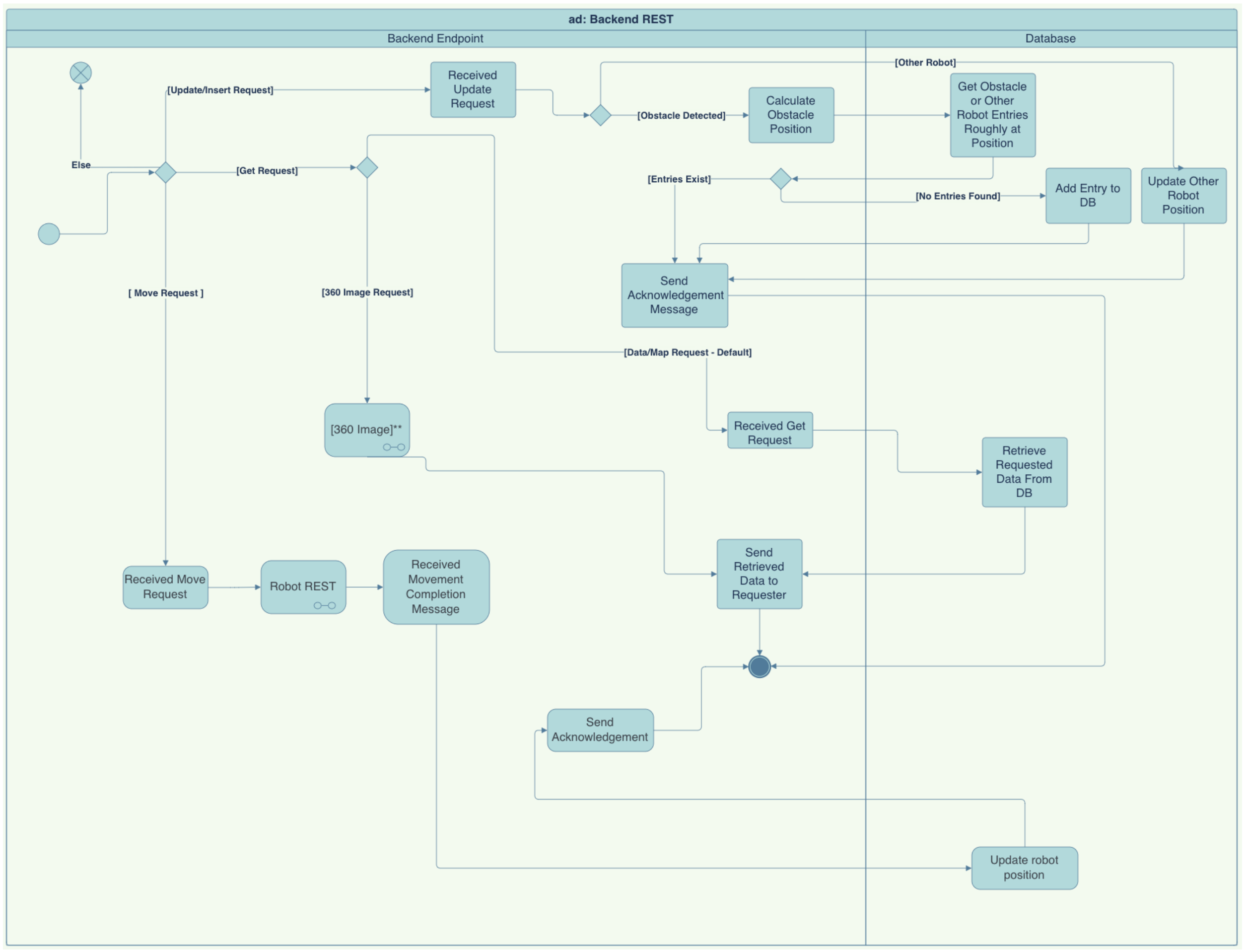

Sd: Robot Movement
:REST Endpoint
:Robot
:Motor
Selected movement type
Activates Motor
opt
[forward]
Turn both motors X rotations forwards*
[backward]
Turn both motors X rotations backwards*
[pivot right]
Turn turn left motor X rotations forwards*
[pivot left]
Turn turn left motor X rotations forwards*
ref
Robot Communication
ref
Update Data
Send Acknowledgement

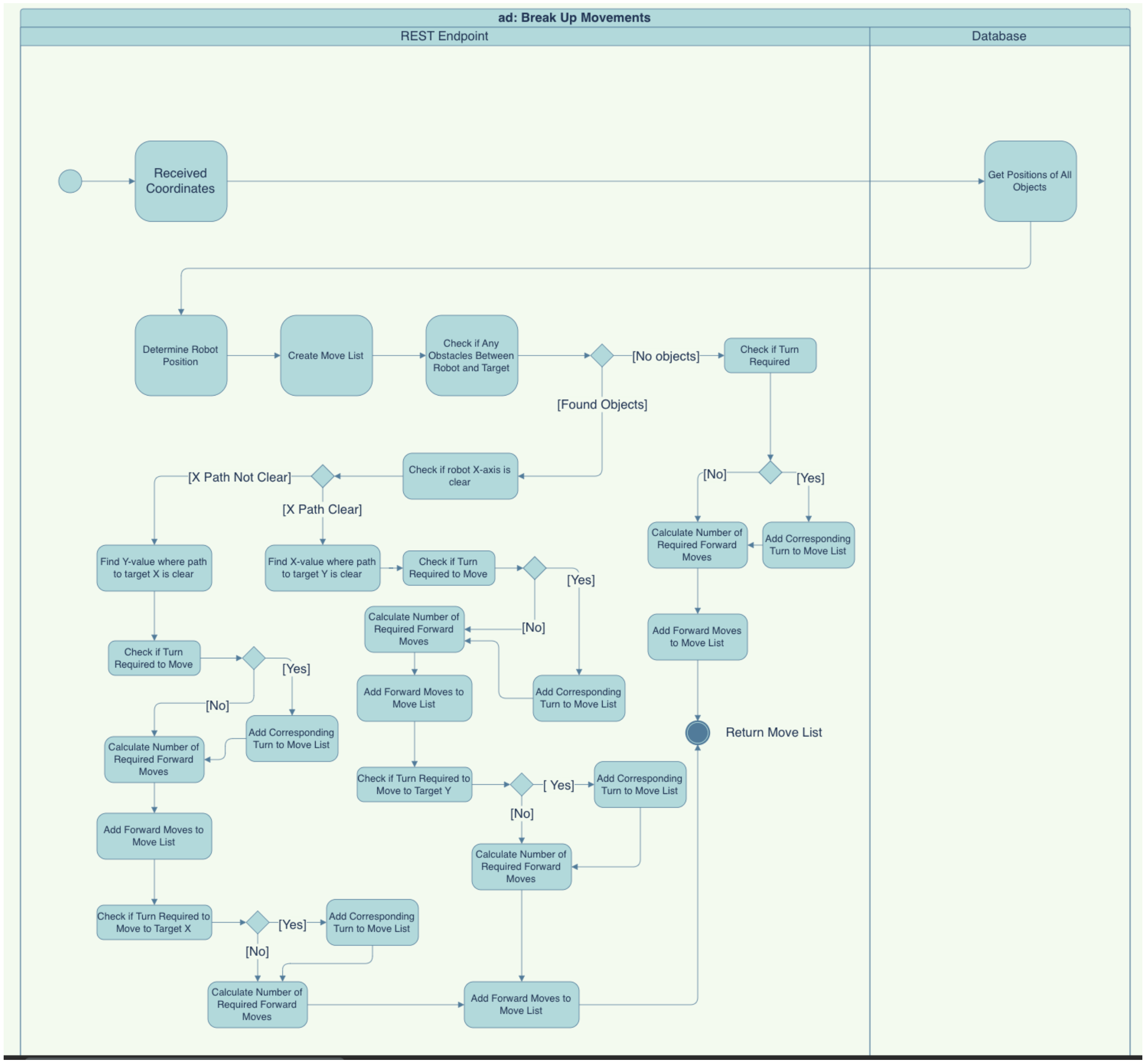

Sd: Stop User Processes
:REST Endpoint
:Robot
Stop Process method called
Ignore all user input until otherwise instructed
return when all movement processes stopped
Sd: Begin Processes
:REST Endpoint
:Robot
Begin Processes method called
Begin listening for user input
return user processes can be started again

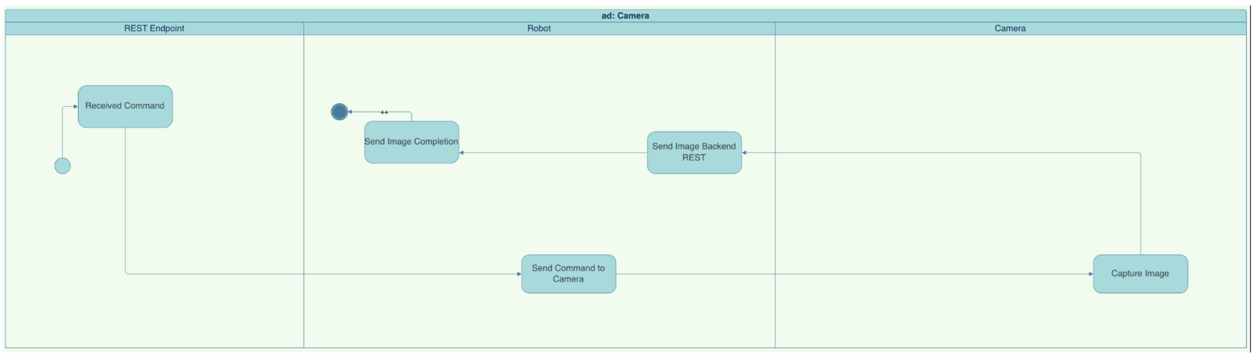

Sd: Update Data
:REST Endpoint
:Backend
:Database
Update request*
Update map/position data**
Acknowledgement
Acknowledgement

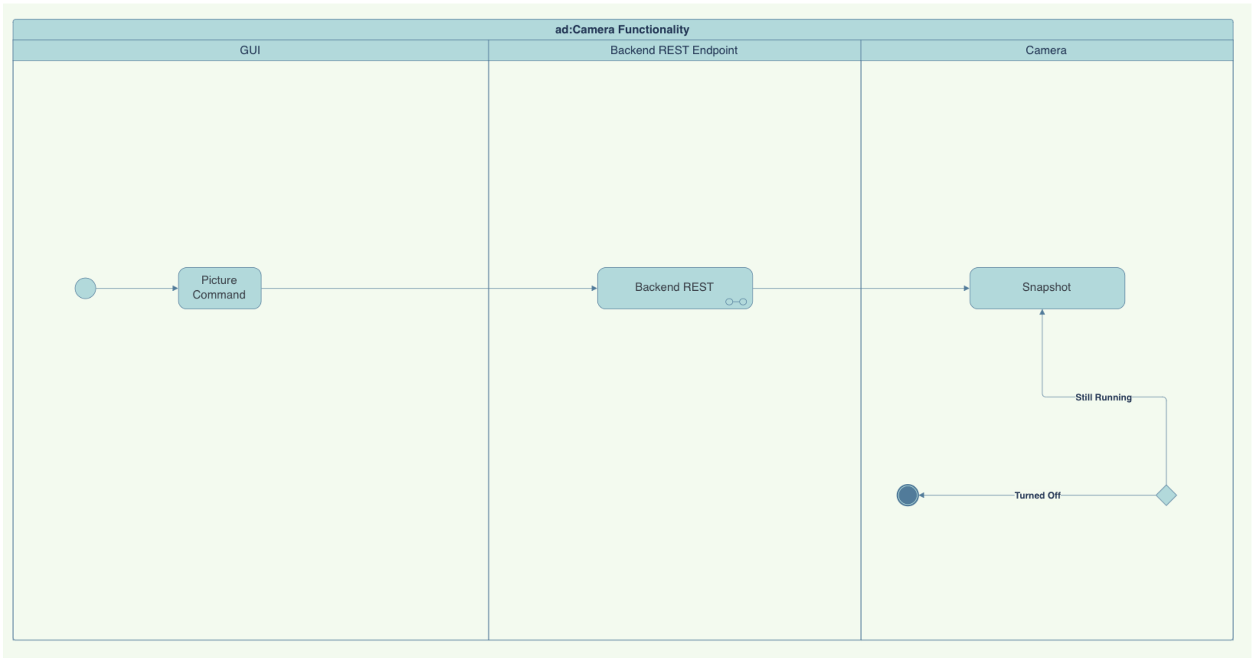
State Machine

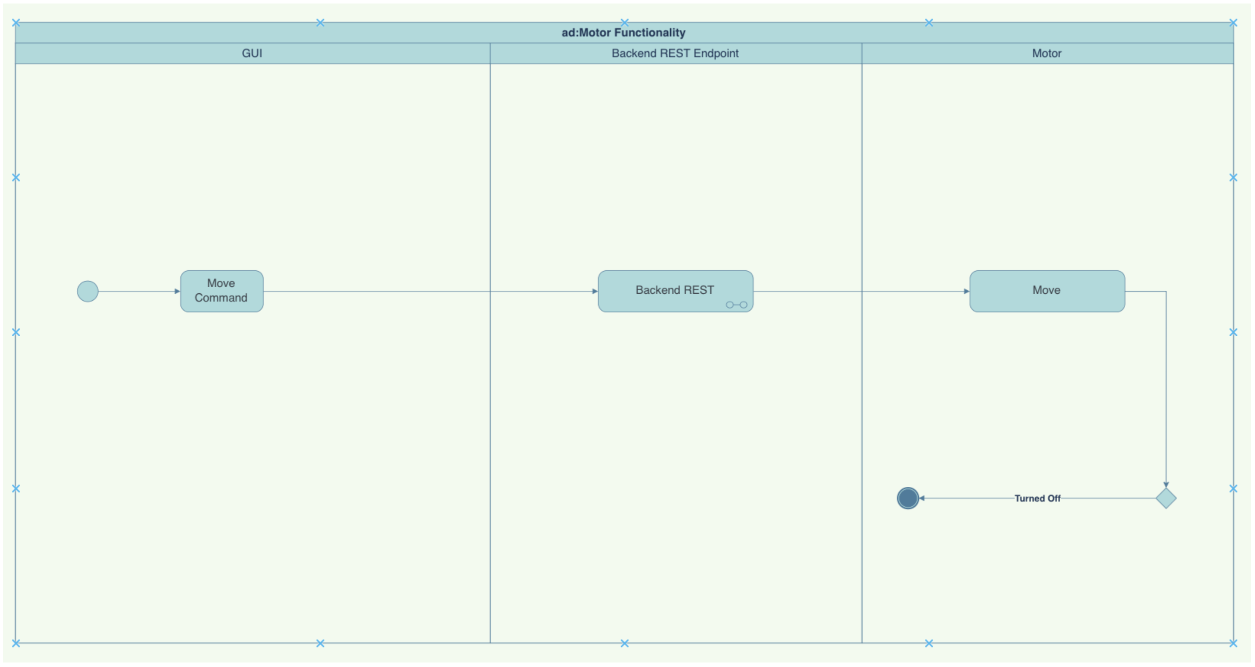

ad: 360 Image
GUI
Backend REST Endpoint
Camera Servo
Camera
Panoramic
**Backend REST: Image Capture
Move 15 degrees (Timer or position TBD)
Image Capture
**Backend REST: Image to Database

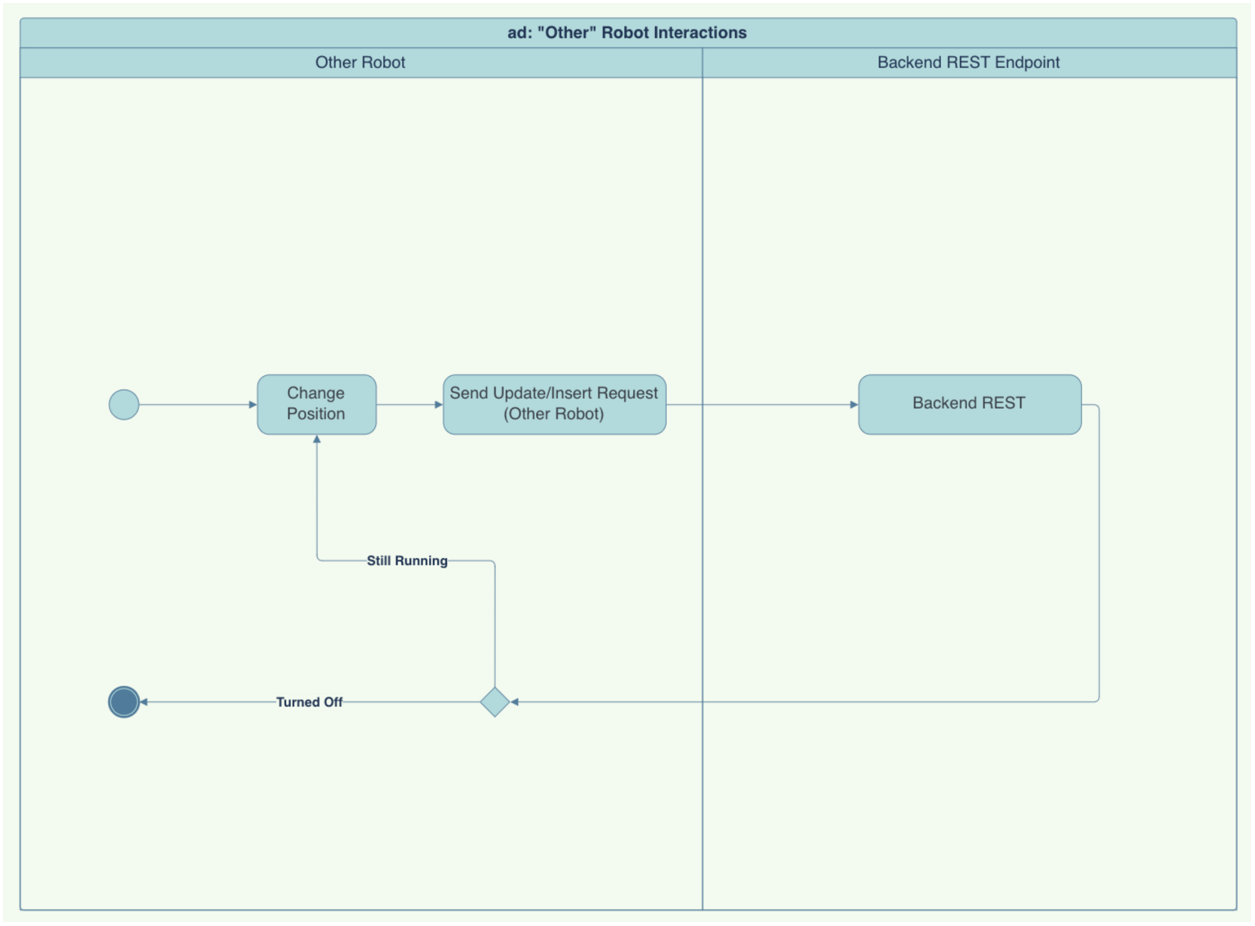


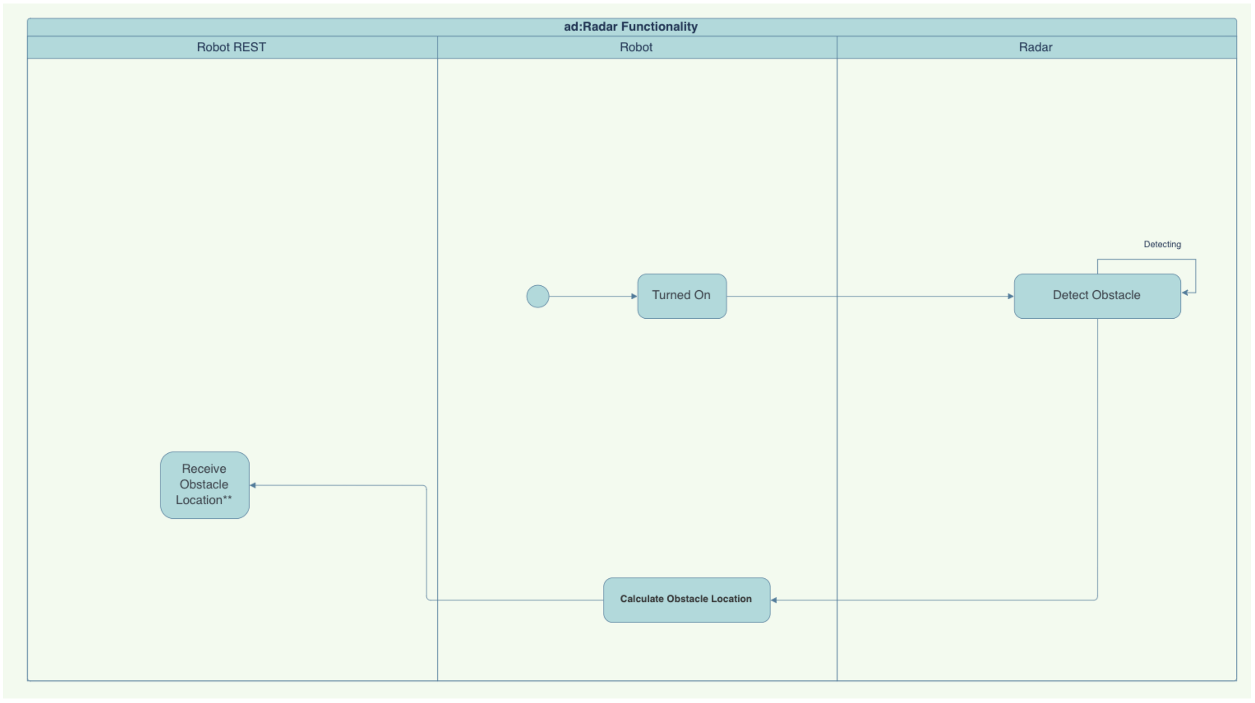

ad: Break Up Movements
REST Endpoint
Database
Received Coordinates
Get Positions of All Objects
Determine Robot Position
Create Move List
Check if Any Obstacles Between Robot and Target
[No objects]
Check if Turn Required
[Found Objects]
Check if robot X-axis is clear
[X Path Not Clear]
[X Path Clear]
[No]
[Yes]
Calculate Number of Required Forward Moves
Add Corresponding Turn to Move List
Find Y-value where path to target X is clear
Find X-value where path to target Y is clear
Check if Turn Required to Move
[Yes]
[No]
Calculate Number of Required Forward Moves
Add Forward Moves to Move List
Check if Turn Required to Move
[Yes]
[No]
Add Corresponding Turn to Move List
Add Forward Moves to Move List
Add Corresponding Turn to Move List
Return Move List
Calculate Number of Required Forward Moves
Check if Turn Required to Move to Target Y
[Yes]
Add Corresponding Turn to Move List
[No]
Calculate Number of Required Forward Moves
Add Forward Moves to Move List
Check if Turn Required to Move to Target X
[Yes]
Add Corresponding Turn to Move List
[No]
Calculate Number of Required Forward Moves
Add Forward Moves to Move List

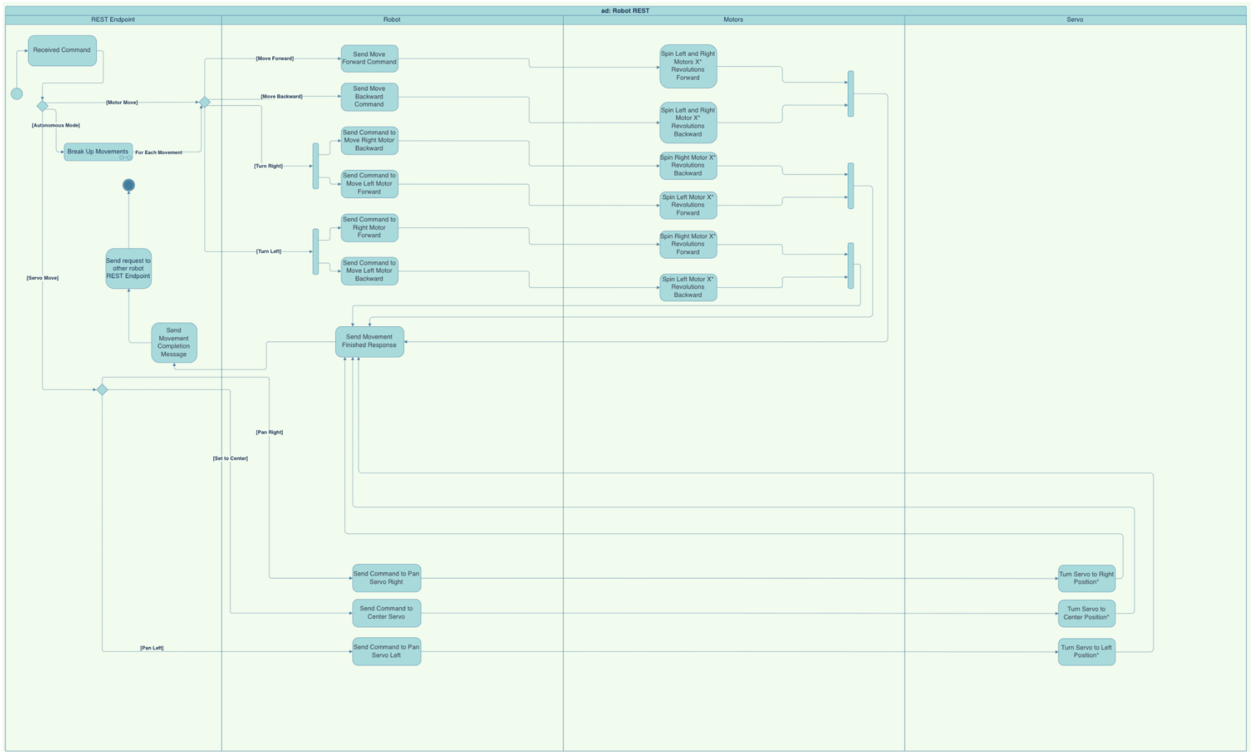

ad: Camera
REST Endpoint
Robot
Camera
Received Command
Send Image Completion
Send Image Backend REST
Send Command to Camera
Capture Image

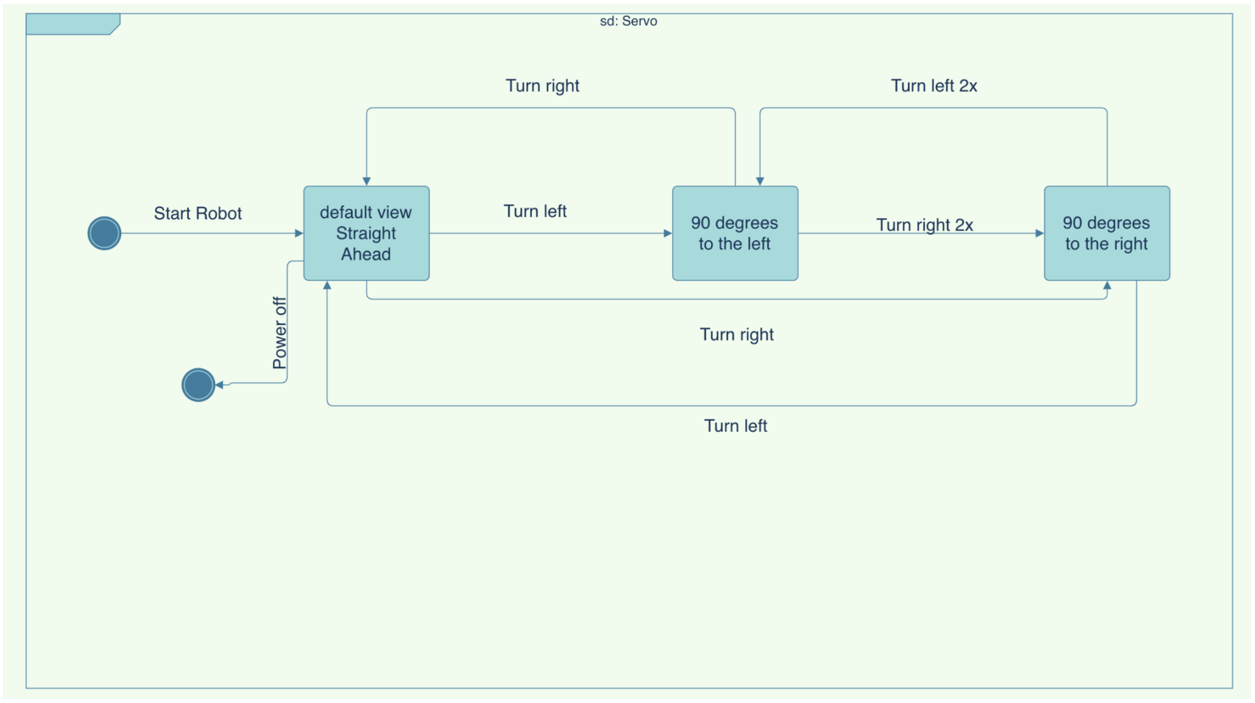

ad:Camera Functionality
GUI
Backend REST Endpoint
Camera
Picture Command
Backend REST
Snapshot
Still Running
Turned Off

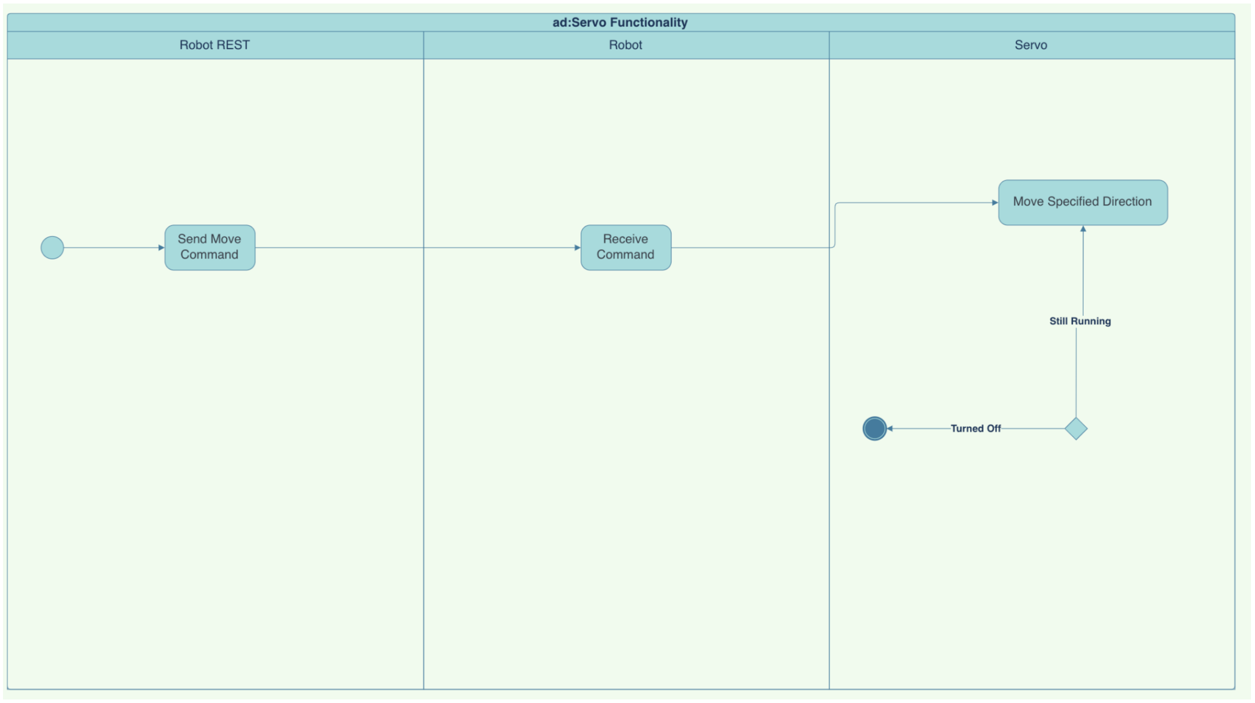

ad:Motor Functionality
GUI
Backend REST Endpoint
Motor
Move Command
Backend REST
Move
Turned Off


ad: "Other" Robot Interactions
Other Robot
Backend REST Endpoint
Change Position
Send Update/Insert Request (Other Robot)
Backend REST
Still Running
Turned Off


ad:Radar Functionality
Robot REST
Robot
Radar
Detecting
Turned On
Detect Obstacle
Receive Obstacle Location**
Calculate Obstacle Location


ad: Robot REST
REST Endpoint
Robot
Motors
Servo
Received Command
[Motor Move]
[Autonomous Mode]
Break Up Movements
For Each Movement
[Servo Move]
Send request to other robot REST Endpoint
Send Movement Completion Message
[Move Forward]
[Move Backward]
[Turn Right]
[Turn Left]
Send Move Forward Command
Send Move Backward Command
Send Command to Move Right Motor Backward
Send Command to Move Left Motor Forward
Send Command to Right Motor Forward
Send Command to Move Left Motor Backward
Send Movement Finished Response
Spin Left and Right Motors X* Revolutions Forward
Spin Left and Right Motor X* Revolutions Backward
Spin Right Motor X* Revolutions Backward
Spin Left Motor X* Revolutions Forward
Spin Right Motor X* Revolutions Forward
Spin Left Motor X* Revolutions Backward
[Pan Right]
[Set to Center]
[Pan Left]
Send Command to Pan Servo Right
Send Command to Center Servo
Send Command to Pan Servo Left
Turn Servo to Right Position*
Turn Servo to Center Position*
Turn Servo to Left Position*

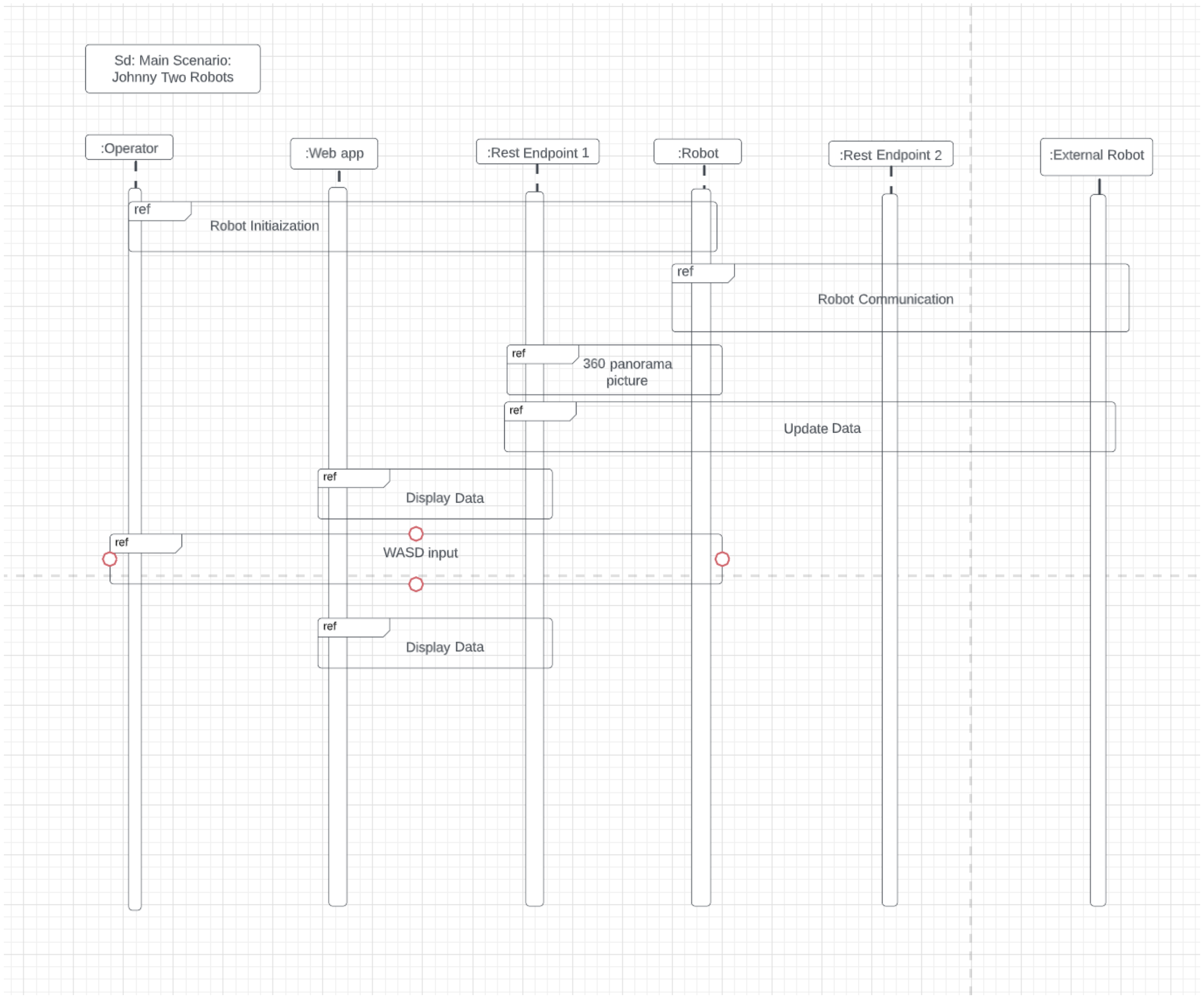

sd: Servo
Start Robot
default view Straight Ahead
Turn left
90 degrees to the left
Turn right 2x
90 degrees to the right
Turn right
Turn left 2x
Turn right
Turn left
Power off

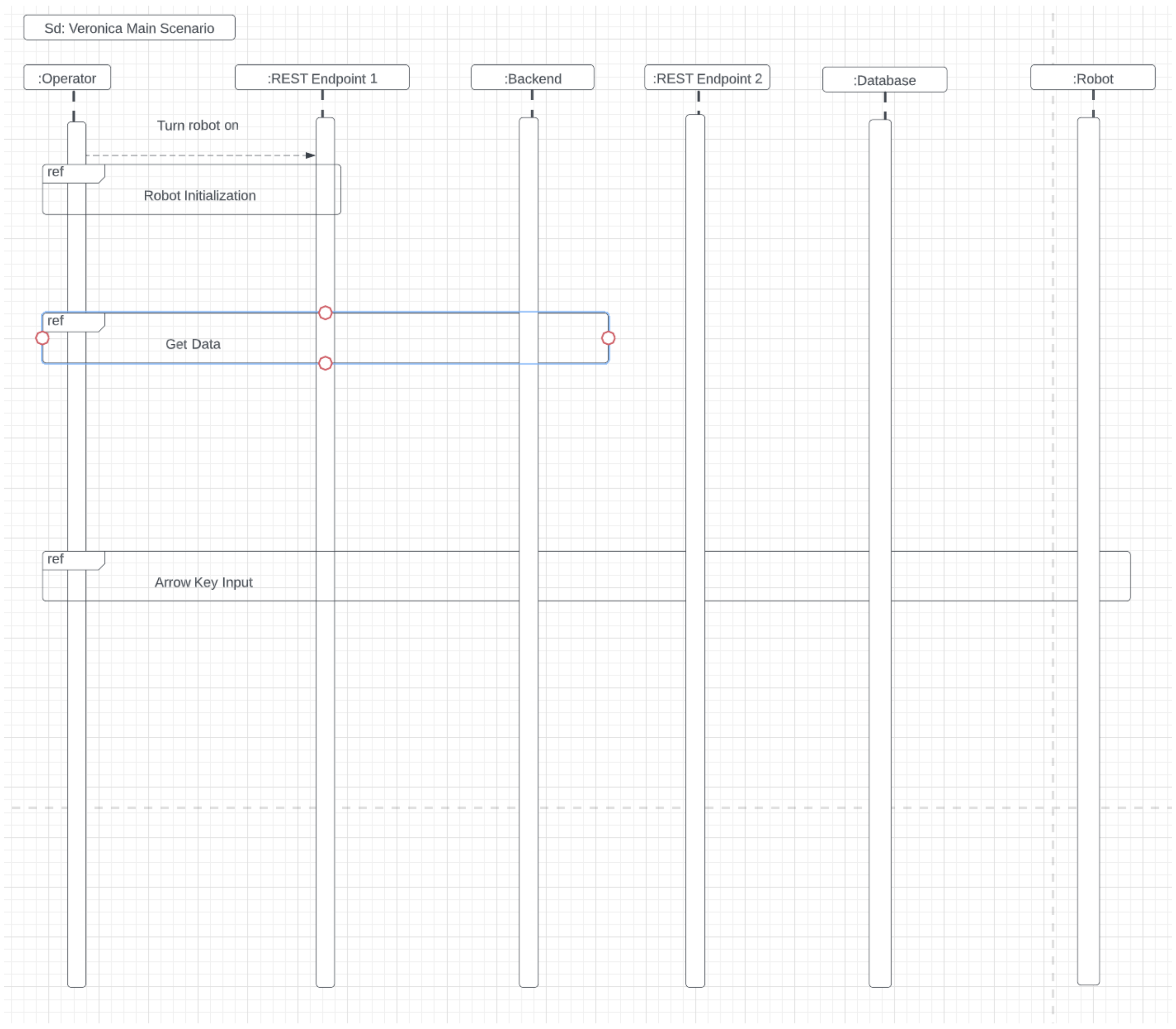


1.3.4 User characteristics

Scenarios (all of these scenarios will be optative, or describing our desired system, compared to an indicative scenario, as we have no software written yet)

These scenarios can be found amongst the UML diagrams here.


Sd: Main Scenario: Johnny Two Robots
:Operator
:Web app
:Rest Endpoint 1
:Robot
:Rest Endpoint 2
:External Robot
ref
Robot Initiaization
ref
Robot Communication
ref
360 panorama picture
ref
Update Data
ref
Display Data
ref
WASD input
ref
Display Data



User Stories

- As a Computer Science teacher, I need the robot to be able to move autonomously
- As a Computer Science teacher, I need the ability to control the robot
- As a user, I want the robot to be easy to control
- As a user, I want the robot to take pictures and panoramas of its surroundings
- As a researcher, I need the system to have descriptive and useful documentation

- As a firefighter, I need to be able to detect weak structures before entering.
- As a robotics competition participant, I want to be able to map out potential positions of other robots
- As a robotics competition participant, I want to be able to switch between web app windows if I am controlling more than one robot

The intended audience is technologically literate people who have experience with robotics.

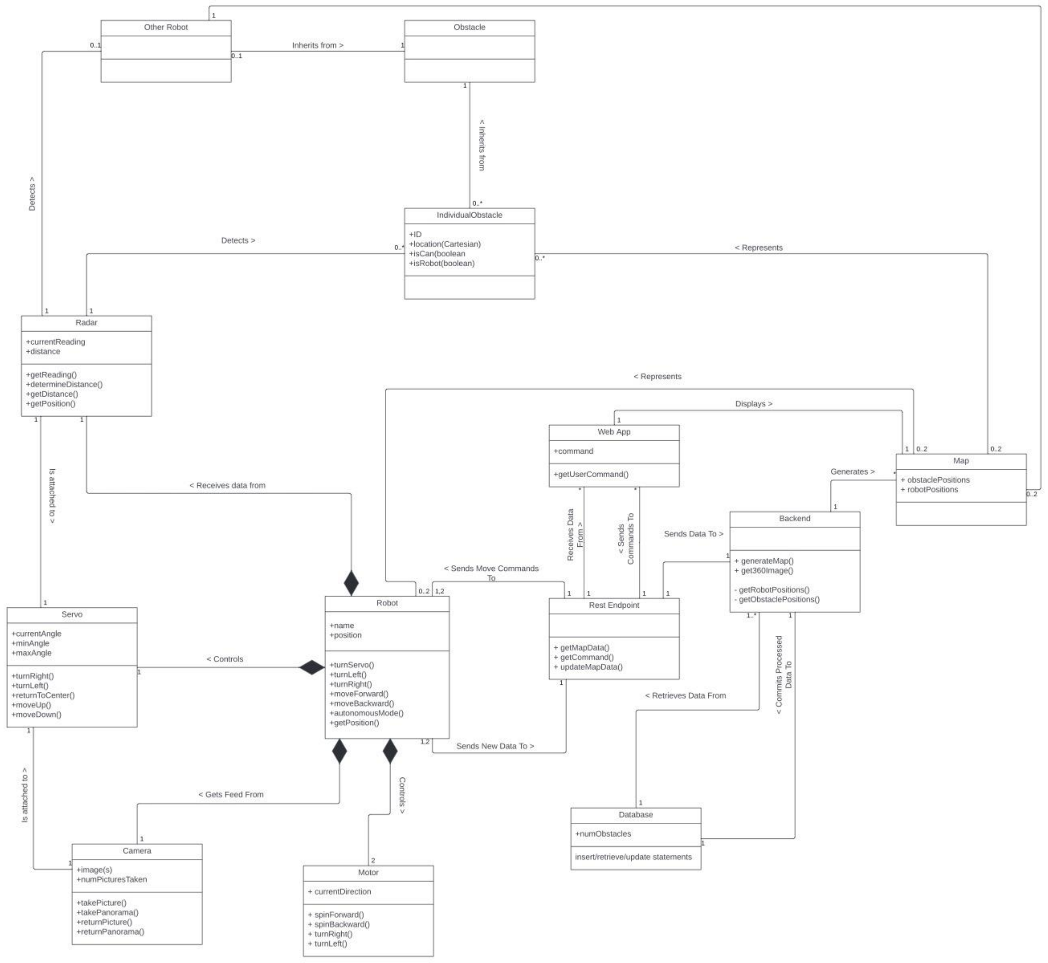
1.3.5 Limitations

We have a deadline of Dec 2, 2022. We are also limited by the abilities of the Raspberry Pi 2 and sensor quality. In addition, the DC motors on the robot can be finicky and imprecise (eg. the bot may struggle to drive in a straight line).

We do not have a budget - we need to use what has been provided by the school.

1.3.6 Class Diagrams

Link to LucidChart



1.4 Definitions

[Include any terms and definitions needed to understand the SRS or the System Under Development (SUD), terms placed here should also be placed in the Appendix. Terms should use

the full name and the general definition of that term, any abbreviations that will be used in the document and the source should be placed in Appendix 5.2]

Term	Definition
Example: *context of use*	*users, tasks, equipment (hardware, software and materials), and the physical and social environments in which a product is used*
System Under Development	The system which is actively being developed.
TBD	To be determined at a later date, the date
NTV	Need to verify; something may not be accurate
UMV	Unmanned vehicle

2. References

[Include citations to external sources and resources in this section. References to other internal documents can be placed here but should also be referenced in the appendix.
Example: ISO/IEC/IEEE 29148.2018, Systems and software engineering
Life cycle processes — Requirements engineering]

UML Diagrams

Flask License

Flask CORS (MIT License)

Python Pillow License

Psycopg2 License

Flask-Session (BSD License)

3. Requirements

[This section should contain all the software requirements at a level of detail sufficient enough to enable designers to design a system, and for testers to test that system, in a way that satisfies the requirements. Each requirement should be perceivable by users, operators, or other external systems. At minimum, the description should include the inputs and outputs of the system, and all functions performed by the system in response to an input or in support of an output. Specific requirements should include the following characteristics:

- *Correct*
- *Unambiguous*
- *Complete*
- *Consistent*
- *Verifiable*
- *Modifiable*
- *Traceable*
- *Ranked for importance and/or stability*
- *uniquely identifiable (usually by numbering)*
- *organized in a way that allows for maximum readability*

The purpose of the requirement is not to dictate design, but rather to guide designers to make the safest, most correct version of the system possible. Do not attempt to build solutions to your written requirements]

3.1 Functions

[Define the fundamental actions that the system must stake in order to accept inputs and generate outputs. It may make sense to organize or partition the functional requirements into sub-functions or sub-processes, do not expect development to mimic this organization. Functional requirements are typically identified using "shall" statements, and some functions that should be included in this section are:

- *Validity checks on the inputs*
  - *"the system shall check the validity of the inputs it receives."*
- *Exact sequence of operations*
  - *"The system shall receive the input, then it shall process the input, and then it shall generate the output."*
- *Responses to abnormal situations including:*
  - *Overflow*
    - *"If the system experiences data overflow, it shall temporarily stop accepting new inputs."*
  - *Communication facilities*
  - *Error Handling and Recovery*

- *Effect of Parameters*
- *Relationships of outputs to inputs*
    - *I/O sequences*
    - *Any formulas used for I/O conversion*

System functions:

System will take user input via a web interface allowing the user to move the robot. The user will click on one of four arrows, which will then send a request to the server, which will convert the request into a move command for the robot to perform. The robot will then move in the direction the user selected.

Other functionalities:

The user will select the panorama picture button. This will send a request to the backend, and then the backend will send the request to the robot. The robot will take a panorama picture and then will send those four files to the backend. The backend will stitch them together and send the fully stitched panorama picture to the frontend for the user to view.

The user will select a location on the map they want the robot to move(not using the arrow keys), using the generate path button. This will then develop a move list and send it to the backend and then send the movelist to the robot for it to perform each move to get to the location specified.

3.2 Performance Requirements

[In measurable terms, specify the numerical requirements of the system. Include static performance requirements such as the number of terminals, simultaneous users, etc. As well as dynamic performance requirements such as the number of tasks able to be completed in a set period of time.]

-The robot appears to be about 20 arcseconds off when attempting to move in a straight line.
-There appears to be a slight lag between clicking the move buttons and the robot actually moving, overall performance is delayed roughly 1-2 seconds per move request.
-There are no more than one user at a time for this robot and only one user is meant to be controlling this robot at a time.
-the system can accomplish one request at a time, such as only moving forward and then turning left.

3.3 Usability Requirements

[Define usability and quality requirements that are measurable in effectiveness, efficiency, satisfaction, and in avoidance of harm that could arise from specific use cases.]

The system is decently effective in its goal to detect obstacles and move. It is accurate to 3-5cm's in determining the distance of an obstacle to its true distance. One should NOT rely on this robot system for precise measurements as it is a very imprecise system and most measurements are off to quite a large degree. The system performs all of its functions as needed, but this is by no means a reliable source of information for real-life measurements and experiments.

3.4 Interface Requirements

[List all inputs and outputs from the system. It should mirror but not repeat the information found in sections 4.2- 4.6. For each defined interface, be sure to include:

- *the name of the item*
- *description of the purpose of the interface*
- *source of input OR output destination*
- *range, accuracy and/or tolerance*
- *units of measurement*
- *timing*
- *I/O relationships*
- *data formats*
- *command formats*

any information included within the I/O.]]

Robot
-Karr
-This is what the system is based on, it performs the actions of detecting obstacles, taking panorama pictures, and movement around the map
-Outputs radar and photo information, inputs move and picture requests
-Not very accurate, cannot drive straight, very imprecise
-Measurements are taken in Millimeters
-Data format: JSON

Server
-Moxie
-This system serves as the middle man between the robot and the front-end. It performs many calculations as well as receiving and sending requests between the robot and the front-end
-It is quite accurate in regards to its calculations and database operations
-Measurements are taken in Millimeters

-Data format: JSON

Front-end
-React
-This system displays all relevant information to the user and allows the user to command the robot what to do
-this system is not too precise, using large block sizes to counteract the imprecision of the robot;s measurements
-Measurements are made in Centimeters
-Data format: JSON

3.5 Logical Database Requirements

[Identify the logical requirements for information that will be place in a database. These requirements should consider and include:

- *Types of information that will be used.*
- *Frequency of use*
- *accessibility*
- *integrity constraints*
- *security*
- *data retention*
- *data entities and their relationships.]*

The database stored coordinates of obstacles and of the other robot(not used). It is used for obstacle avoidance as it would be checked to determine if an obstacle was in the robots path.

3.6 Design Constraints

- Robot must be placed in the middle of the usable area and facing East to start
- Surface should be flat
- Internet connection required

3.7 Software System Attributes

[For each of the attributes of the software system (Reliability, Availability, Security, Maintainability, Portability, etc.), list the factors that will establish functionality or stability. For example, when establishing requirements for the Security attribute, you may include one that restricts communication between two areas of the program and another.]
Reliability:

-Robot will take in distance measurements and move in the relative direction specified by the user
Availability:
-As long as the robot has a source of power, it will perform operations to relatively the same consistency.
Security:
-The robot only communicates with the server, as well as the front-end only communicates with the server, following security patterns in line with MVC.
Maintainability:
-All code is open source allowing for full transparency and maintainability. The system is also well documented.
Portability:
-This system has specific features that would be difficult to emulate without some major changes on other systems. This system is not meant to be easily ported.

3.8 Supporting Information

[Add any additional information needed to understand the SRS, include things like background information, problem descriptions, packaging instructions for code, sample input/output formats, etc.]
N/A

Backend-Rest

## Creating Docker Image and Testing Locally
Make sure you have Docker installed!

To get testing:
1. Run `docker build -t flask-app-backend . `
2. Run `docker run -p 9823:5000 -d flask-app-backend`
3. Hit `http://localhost:9823/logs?password=<password>&istest=False` in either Postman or your
browser. If it works, everything built correctly!

## API Specification

### /initialize

#### GET
For the robot to hit so its public IP can be stored in the `robot_ip` session
variable.

- Returns
  - `Got IP` (string value)
- URL Parameters
  - None

### /checkip

#### GET
Returns the IP of the robot stored in the session, if available.

- Returns
  - Stored robot public IP as a string
  - Empty string if unavailable
- URL Parameters
  - None

### /generatepano

Tells the robot to take a panoramic view. Images are received as a `.zip`,
are unzipped into `panoramic-images/received-data/...` and then the images are copied out of
the received file structure into the root of `panoramic-images`.

#### GET

- Returns
  - `Panoramic pictures taken and saved` (string)
- URL Parameters
  - None

### /panoramic

Get a stitched together panoramic of pictures the robot takes
facing North, South, East and West. If there are no pictures to stitch, a black
square picture is returned.

#### GET
- Returns

  - Image of type `.png`
- URL Parameters
  - None

### /current_view

Get the latest image saved from the robot's PoV. If there are no pictures to stitch, a black square picture is returned.

#### GET
- Returns
  - Image of type `.png`
- URL Parameters
  - None

#### GET
- Returns
  - Image of type `.png`
- URL Parameters
  - None

### /clear_obstacles

#### GET

Deletes all obstacles for the given map id. Returns what remains for that map - should be the robot record plus the ghost obstacle with matching coordinates.

- Returns
  - JSON object of a list of MapObjects
  - Sample JSON:
  ```json
  [
      {
          "map_id": 1,
          "object_type": "Can",
          "location": [
  ```

```
            5,
            5
        ],
        "created_at": "2022-11-12T00:43:29.532003",
        "direction": null,
        "obj_id": 1
    },
    {
        "map_id": 1,
        "object_type": "OurRobot",
        "location": [
            5,
            5
        ],
        "created_at": "2022-11-12T00:43:29.532003",
        "direction": "E",
        "obj_id": 1
    }
]
```

- URL Parameters
  - *objtype*
    - The type of MapObject to be returned. Valid values are `Can`, `OtherRobot` and `OurRobot`
  - *istest*
    - If this is a local endpoint meant to use a local Docker database (`True` or `False`). Optional, defaults to `False`
  - *password*
    - Password for `flaskuser`.
  - *moxie*
    - Connect to the Docker DB on Moxie (`True` or `False`). Optional, defaults to `False`.

### /mapdata

#### GET

- Returns
  - JSON object of a list of MapObjects

- Sample JSON:
```json
[
  {
    "map_id": 1,
    "object_type": "Can",
    "location": [
      7,
      4
    ],
    "created_at": "2022-11-12T01:13:27.631719",
    "direction": null,
    "obj_id": 3
  },
  {
    "map_id": 1,
    "object_type": "Can",
    "location": [
      5,
      5
    ],
    "created_at": "2022-11-12T00:43:29.532003",
    "direction": null,
    "obj_id": 1
  },
  {
    "map_id": 1,
    "object_type": "OurRobot",
    "location": [
      5,
      5
    ],
    "created_at": "2022-11-12T00:43:29.532003",
    "direction": "E",
    "obj_id": 1
  }
]
```
- URL Parameters
  - *mapid*

  - Id of the Map the returned objects are associated with.
 - *objtype*
  - The type of MapObject to be returned. Valid values are
  `Can`, `OtherRobot` and `OurRobot`
 - *istest*
  - If this is a local endpoint meant to use a local
  Docker database (`True` or `False`). Optional, defaults to `False`
 - *password*
  - Password for `flaskuser`.
 - *moxie*
  - Connect to the Docker DB on Moxie (`True` or `False`).
  Optional, defaults to `False`.

#### POST
Creates a new Map and sets its id to the session variable `current_map_id`.
- Returns
 - `New Map ID: <Map Id>`
- URL Parameters
 - None

### /move

#### GET
Moves the robot in the specified direction, updates its location as well as
saves any new obstacles detected. Will include a "ghost" obstacle whose coordinates match the robot's

- Returns
 - JSON object of a list of MapObjects
 - Sample JSON:
 ```json
 [
    {
       "map_id": 1,
       "object_type": "Can",
       "location": [
          7,
          4
       ],
       "created_at": "2022-11-12T01:13:27.631719",
 ```

```
        "direction": null,
        "obj_id": 3
    },
    {
        "map_id": 1,
        "object_type": "Can",
        "location": [
            5,
            5
        ],
        "created_at": "2022-11-12T00:43:29.532003",
        "direction": null,
        "obj_id": 1
    },
    {
        "map_id": 1,
        "object_type": "OurRobot",
        "location": [
            5,
            5
        ],
        "created_at": "2022-11-12T00:43:29.532003",
        "direction": "E",
        "obj_id": 1
    }
]
```

- URL Parameters
  - *movekey*
    - Key pressed for movement. Valid values are `W` (move forward), `A` (turn left), `S` (move backward) or `D` (turn right).
  - *istest*
    - If this is a local endpoint meant to use a local Docker database (`True` or `False`). Optional, defaults to `False`
  - *password*
    - Password for `flaskuser`.
  - *moxie*
    - Connect to the Docker DB on Moxie (`True` or `False`). Optional, defaults to `False`.

### /testrobotconnect

#### GET
Checks to see if the robot is reachable by sending a request to its
`/coffee` endpoint.

- Returns
  - ● - Success:
    - `Test connect to robot successful!` (string)
  - Failure:
    - `Could not reach robot.` (string)
- URL Parameters
  - None

### /logs

#### GET
Returns a list of JSON-formatted logs from the database.

- Returns
  - Success (sample JSON):
  ```json
  [
    {
      "created_at": "2022-11-04T14:05:32.478150",
      "origin": "app.py",
      "message": "invalid integer value \"localhost\" for connection option \"port\"\n",
      "log_type": "Error",
      "stack_trace": "",
      "id": 10
    },
    {
      "created_at": "2022-11-04T14:07:12.071802",
      "origin": "app.py",
      "message": "Logs retrieved from /logs GET",
      "log_type": "Event",
      "stack_trace": "",
      "id": 11
    }
  ]
  ```

~~~
 - Failure:
   - `An error occurred - see logs` (string)
- URL Parameters
 - *istest*
   - If this is a local endpoint meant to use a local
   Docker database (`True` or `False`). Optional, defaults to `False`
 - *password*
   - Password for `flaskuser`.
 - *moxie*
   - Connect to the Docker DB on Moxie (`True` or `False`).
   Optional, defaults to `False`.

### /autonomous

#### POST
Receives a set of Dijkstra-calculated coordinates to be translated into movements
to send to the robot.  Will return the current map objects, including a "ghost" obstacle
whose coordinates match the robot's.  This ghost is required for the frontend to execute its
Dijkstra command.

- Request Body Structure
 ```json
   {
     "Coordinates":
      [
        [15, 1],
        [15, 2],
        [15, 3],
        [15, 4],
        [15, 5],
        [15, 6],
        [15, 7],
        [15, 8],
        [15, 9]
      ]
   }
 ~~~
- Returns

- JSON object of a list of MapObjects
- Sample JSON:

```json
[
  {
    "map_id": 1,
    "object_type": "Can",
    "location": [
      7,
      4
    ],
    "created_at": "2022-11-12T01:13:27.631719",
    "direction": null,
    "obj_id": 3
  },
  {
    "map_id": 1,
    "object_type": "Can",
    "location": [
      5,
      5
    ],
    "created_at": "2022-11-12T00:43:29.532003",
    "direction": null,
    "obj_id": 1
  },
  {
    "map_id": 1,
    "object_type": "OurRobot",
    "location": [
      5,
      5
    ],
    "created_at": "2022-11-12T00:43:29.532003",
    "direction": "E",
    "obj_id": 1
  }
]
```

- URL Parameters

 - *istest*
   - If this is a local endpoint meant to use a local
   Docker database (`True` or `False`). Optional, defaults to `False`
 - *password*
   - Password for `flaskuser`.
 - *moxie*
   - Connect to the Docker DB on Moxie (`True` or `False`).
   Optional, defaults to `False`.

### /move

#### GET
Moves the robot in the specified direction, updates its location as well as
saves any new obstacles detected.

- Returns
  - JSON object of a list of MapObjects
  - Sample JSON:
  ```json
  [
      {
          "map_id": 1,
          "object_type": "Can",
          "location": [
              31,
              25
          ],
          "created_at": "2022-11-12T01:13:27.631719",
          "direction": null,
          "obj_id": 3
      },
      {
          "map_id": 1,
          "object_type": "OurRobot",
          "location": [
              26,
              25
          ],
          "created_at": "2022-11-12T00:43:29.532003",
  ```

        "direction": "E",
        "obj_id": 1
      }
  ]
  ~~~
- URL Parameters
  - *movekey*
    - Key pressed for movement.  Valid values are `W` (move forward), `A` (turn left), `S` (move backward) or `D` (turn right).
  - *istest*
    - If this is a local endpoint meant to use a local Docker database (`True` or `False`). Optional, defaults to `False`
  - *password*
    - Password for `flaskuser`.
  - *moxie*
    - Connect to the Docker DB on Moxie (`True` or `False`). Optional, defaults to `False`.

### /other_robot

#### POST
Endpoint for the other robot team to send their updated position and current radar reading to.

- Request Body Structure
  ```json
  {
      "location": [
          3,
          2
      ],
      "orientation": "E",
      "radar": 48
  }
  ~~~
    - `location` should have both coordinate values in meters
    - `radar` is measured in centimeters
- Returns
  - JSON object of a list of MapObjects
  - Sample JSON:
  ```json
  ```

```
[
  {
    "map_id": 1,
    "object_type": "OurRobot",
    "location": [
      14,
      1
    ],
    "created_at": "2022-11-12T00:43:29.532003",
    "direction": "E",
    "obj_id": 1
  },
  {
    "map_id": 1,
    "object_type": "Can",
    "location": [
      35,
      20
    ],
    "created_at": "2022-11-15T19:31:31.705408",
    "direction": null,
    "obj_id": 10
  },
  {
    "map_id": 1,
    "object_type": "OtherRobot",
    "location": [
      30,
      20
    ],
    "created_at": "2022-11-15T19:31:31.643092",
    "direction": "E",
    "obj_id": 9
  }
]
```

- URL Parameters
  - *istest*
    - If this is a local endpoint meant to use a local Docker database (`True` or `False`). Optional, defaults to `False`

- *password*
  - Password for `flaskuser`.
- *moxie*
  - Connect to the Docker DB on Moxie (`True` or `False`).
  Optional, defaults to `False`.

Client

# Getting Started with Robot Radar GUI (Group C)

This project was bootstrapped with [Create React App](https://github.com/facebook/create-react-app).
<<<<<<< HEAD
In addition, we have implemented the [Material UI](https://mui.com/) frontend component libraries, for each structured card amd icon. The Pannellum library was used for panoramic generation. The radar gui design and Dijkstra visualization was inspired by the open source [Path Finder Visualization Tool](https://github.com/PrudhviGNV/pathFinderVisualizer). Although functionally similiar, the implementation of React functional component and mapping functions were within the bounds of our own project work.

## How to Use

#### Manual Update Button
This button MUST be clicked twice.

Required after every move command in order to plot the current location of the robot and obstacles in the field. The Forward and Reverse arrow commands are the only commands which collect radar data, therfor, a left or right movement must be followed by a forward or reverse command in order to capture surrounding obstacles and location data.

#### Get Logs Button
Returns a list of all logs, with the most recent rendered at the top of the log card.

#### Generate Panoramic Button
This button sends a POST request to the server indicating that the robot must move and capture images, which are stithced together by the server.

#### Refresh Icon Button
This button retrieves the generated panoramic image. NOTE: The page must be refreshed in order to retrieve a panoramic image
generated after the first generation.

## Movement Controls
The arrow buttons control basic up, down left, right movement in conjunction with the location where they rest. In addition, there exists an event listener which permits the WASD keyboard controls.

## Radar Functionality

#### Generate Path Button
Generate Path, displays and logs the current block path from the robot to the pre-defined target. One must ALWAYS run the generate path button before the execute path button.

#### Execute Path Button
Sends the Dijkstra coordinate array to the server. These are then into physical robot movement commands,
and the robot will move to the pre-defined target. Due to compounding movement errors, it will likely not end up exactly in that spot.

After this, you must hit the System Reset button

#### System Reset Button
Refreshes the page. Must be used after Execute Path has been used. Hit Manual Update button twice to refresh the grid.
The robot will not show up as it is currently at the target button - move it and it will show again after a Manual Update.

## Use Sequence

Runs the app if and only if the robot is on.
Open [Robot Radar Client](http://moxie.cs.oswego.edu:36001/home) to view it in your browser.

## Running locally

Remember to cd into the client directory before trying to run the app.

Use `npm start` to start the app.

## Building to Docker & Deployment

Run `docker build -t robot-radar-client .` to build the docker image.

To run the container ensure that the port mapping maps the local host port `3000` to the desired target port.

The following command encompasses the required instructions.

`docker run -dp 36001:3000 robot-radar-client`

#### Deploying to Moxie General Instructions
Save your image on your local machine:
`docker save robot-radar-client > robot-radar-client.tar`
Upload tar to your remote server:
`scp robot-radar-client.tar studentID@moxie.cs.oswego.edu:.`
Load image on your remote machine:
`ssh studentID@moxie.cs.oswego.edu`
`docker load < robot-radar-client.tar`

Run a new container

`docker run -dp 36001:3000 robot-radar-client`
o
=======
In addition, we have implimented the [Material UI](https://mui.com/) frontend component libraries, for each structured card amd icon. The Pannellum library was used for panoramic generation. The radar gui design and djikstra visualization was inspired by the open source [Path Finder Visualization Tool](https://github.com/PrudhviGNV/pathFinderVisualizer). Although functionally similiar, the implimentation of react functional components, and mapping functions were within the bounds of our own project work.

## How to Use

#### Get Logs Button
Returns a list of all logs, with the most recent rendered at the top of the log card.

#### Generate Panoramic Button
This button sends a POST request to the server indicating that the robot must move and capture images, which are stithced together by the server.

#### Refresh Icon Button
This button retrieves the generated Panormaic image. NOTE: The page must refresh in order to re-send a panoramic image request.

## Movement Controls
The arrow buttons control basic up, down left, right movement in conjunction with the location where they rest. In addition, there exists an event lister which permits the WASD keyboard controls.

## Radar Functionality

#### Refresh Button
This button clears the field after a a djikstra visualization occurs.

#### Manual Update Button
This button is required after every move command in order to plot the current location of the robot and obstacles in the field. The Forward and Reverse arrow commands are the only commands which collect radar data, therfor, a left or right movement must be followed by a forward or reverse command in order to capture surrounding obstacles and location data.

#### Generate Path Button
Generate Path, displays and logs the current block path from the robot to the target. One must ALWAYS run the generate path button before the execute path button.

#### Execute Path Button
This button send the Djikstra coordinate array to the server which translate them into physical robot movement commands.

## Use Sequence

Runs the app if and only if the robot is on.
Open [Robot Radar Client](http://moxie.cs.oswego.edu:36001/home) to view it in your browser.

## Running locally

***Remember to cd into the client directory before trying to run the app.
use `npm start` to start the app.

## Building to Docker & Deployment

Run `docker build -t robot-radar-client .` to build the docker image.

To run the container ensure that the port mapping maps the local host port `3000` to the desired target port.

The following command encompasses the required instructions.

`docker run -dp 36000:3000 robot-radar-client`

#### Deploying to Moxie General Instructions
Save your image on your local machine:
`docker save robot-radar-client > robot-radar-client.tar`
Upload tar to your remote server:
`scp robot-radar-client.tar studentID@moxie.cs.oswego.edu:.`
Load image on your remote machine:
`ssh studentID@moxie.cs.oswego.edu`
`docker load < obot-radar-client.tar`

Run a new container

`docker run -dp 36000:3000 robot-radar-client`

>>>>>>> c22702c (Post deployment: Updated ReadME)

4. Verification

[List all inputs and outputs from the system. It should mirror but not repeat the information found in sections 3.1- 3.8. For each defined interface, be sure to include:

- *the name of the item*

- *description of the purpose of the interface*
- *source of input OR output destination*
- *range, accuracy and/or tolerance*
- *units of measurement*
- *timing*
- *I/O relationships*
- *data formats*
- *command formats*
- *any information included within the I/O.]*

4.1 Functions

[See sections 4.0, 3.1 for specific directions about what outputs should be included here.]

System functions:

System will take user input via a web interface allowing the user to move the robot. The user will click on one of four arrows, which will then send a request to the server, which will convert the request into a move command for the robot to perform. The robot will then move in the direction the user selected.

Other functionalities:

The user will select the panorama picture button. This will send a request to the backend, and then the backend will send the request to the robot. The robot will take a panorama picture and then will send those four files to the backend. The backend will stitch them together and send the fully stitched panorama picture to the frontend for the user to view.

The user will select a location on the map they want the robot to move(not using the arrow keys), using the generate path button. This will then develop a move list and send it to the backend and then send the movelist to the robot for it to perform each move to get to the location specified.

4.2 Performance Requirements

[See sections 4.0, 3.2 for specific directions about what outputs should be included here.]

-The robot appears to be about 20 arcseconds off when attempting to move in a straight line.
-There appears to be a slight lag between clicking the move buttons and the robot actually moving, overall performance is delayed roughly 1-2 seconds per move request.
-There are no more than one user at a time for this robot and only one user is meant to be controlling this robot at a time.

-the system can accomplish one request at a time, such as only moving forward and then turning left.

4.3 Usability Requirements

[See sections 4.0, 3.3 for specific directions about what outputs should be included here.]

The system is decently effective in its goal to detect obstacles and move. It is accurate to 3-5cm's in determining the distance of an obstacle to its true distance. One should NOT rely on this robot system for precise measurements as it is a very imprecise system and most measurements are off to quite a large degree. The system performs all of its functions as needed, but this is by no means a reliable source of information for real-life measurements and experiments.

4.4 Interface Requirements

[See sections 4.0, 3.4 for specific directions about what outputs should be included here.]

Robot
-Karr
-This is what the system is based on, it performs the actions of detecting obstacles, taking panorama pictures, and movement around the map
-Outputs radar and photo information, inputs move and picture requests
-Not very accurate, cannot drive straight, very imprecise
-Measurements are taken in Millimeters
-Data format: JSON

Server
-Moxie
-This system serves as the middle man between the robot and the front-end. It performs many calculations as well as receiving and sending requests between the robot and the front-end
-It is quite accurate in regards to its calculations and database operations
-Measurements are taken in Millimeters
-Data format: JSON

Front-end
-React

-This system displays all relevant information to the user and allows the user to command the robot what to do
-this system is not too precise, using large block sizes to counteract the imprecision of the robot;s measurements
-Measurements are made in Centimeters
-Data format: JSON

4.5 Logical Database Requirements

[See sections 4.0, 3.5 for specific directions about what outputs should be included here.]

The database stored coordinates of obstacles and of the other robot(not used). It is used for obstacle avoidance as it would be checked to determine if an obstacle was in the robots path.

-Database Schema is located in our documentation

4.6 Design Constraints

[See sections 4.0, 3.6 for specific directions about what outputs should be included here.]

- Robot must be placed in the middle of the usable area and facing East to start
- Surface should be flat
- Internet connection required

4.7 Software System Attributes

[See sections 4.0, 3.7 for specific directions about what outputs should be included here.]

-Robot will take in distance measurements and move in the relative direction specified by the user
Availability:
-As long as the robot has a source of power, it will perform operations to relatively the same consistency.
Security:
-The robot only communicates with the server, as well as the front-end only communicates with the server, following security patterns in line with MVC.
Maintainability:
-All code is open source allowing for full transparency and maintainability. The system is also well documented.

Portability:
-This system has specific features that would be difficult to emulate without some major changes on other systems. This system is not meant to be easily ported.

4.8 Supporting Information

[See sections 4.0, 3.8 for specific directions about what outputs should be included here.]

N/A

5. Appendix A – Tailoring Policies

5.1 Assumptions and dependencies

[Identify any and all factors that may impact the implementation and execution of the requirements written below. These factors do not add a constraint but may impact development if they are changed. Example: a major update to an operating system(OS) on which the SUD is intended to run impacts the implementation of one of the core features. The version of the OS that the system had intended to run on should be listed in this section.]

5.2 Acronyms and Abbreviations

Term	Definition	Abbreviation	Source (if applicable)
Example: context of use	*users, tasks, equipment (hardware, software and materials), and the physical and social environments in which a product is used*	*COU*	*[SOURCE: ISO/IEC 25000:2014, 4.2]*
System Under Development	The system which is actively being developed.	SUD	

5.3 Tailoring Policies

Tailoring is not a requirement to bring the document into compliance with the standards set by IEEE 29148-2018. Tailoring should only occur when conformance to the standard is not possible or practical. The act of tailoring is the modification and/or removal of one of the content sections outlined in this document, adding additional information items for organization is not considered tailoring. Tailoring should only occur when factors or circumstances:

- surround an organization that is using the document
- influence a project using this document to meet an agreement
- reflect the needs of an organization.

When tailoring the document, the following activities shall be implemented:

- Identify and document the circumstances that may influence tailoring.
    - novelty, size and complexity
    - stability of operating environments
    - variety in operating environments
    - starting date and duration
    - emerging technology
    - availability of services of enabling systems
    - other standards with which the document needs to conform.
- Identify and get input from all parties impacted by the tailoring process.
    - Such as stakeholders, contributors, and other interested parties
- Delete the information contents that require tailoring.

5.4 User Personas

Persona for Teacher

Johnny (whose nickname is Bastian for some reason), age 35, is a college professor at SUNY Oswego, an average sized city on Lake Ontario. He was born in Hamburg, Germany, his father runs an ice cream shop and his mother is a nurse at a hospital. He has his PHD degree in Computer Science and is a trained Software Engineer.

Johnny's experience with Computer Science and software engineering make him quite capable of utilizing the robot for his research needs. He believes that this software will assist in his research pertaining to autonomous systems, and how users would interact with the robot.

Persona for Firefighter

Jim, age 30, is a firefighter for his local municipality in Syracuse, NY. He is responsible for search-and-rescue operations. He was born in Watertown, NY, his father is a mechanic and his mother is a nurse at a hospital. He has his high school degree and training from the Syracuse Fire Department Training.

Jim's proper training with UMV system controls and UMV operation, through the firefighter training course, adequately allows him to use this UMV. He believes that this software will assist insearch and rescue operations.

Persona for Robotics Competition Participant

Veronica, age 26, is from Auburn, NY and has a passion for robotics since she was in eighth grade. Professionally, she's a software developer, but her main love is working on robotics hobby projects. She believes she can now move up a level and enter robotics competitions (similar to the famed Battlebots one) with her skills. When she's not at work or building robots, she's a loving mother to two cats, spends her weekends training for e-sports tournaments and often visits her adoptive parents who live next door.

Veronica's very interested in any new technologies that can give her an edge in the competition. She believes that radar detection capabilities paired with the 360 degree map provided by the robot radar system will help her detect obstacles and enemy robots. She is also interested in the communication capabilities in the event she is able to deploy multiple robots.

Persona for Field Soldier

Franklin, age 33, is a Junior battlefield soldier, part of a small military unit that has been deployed to Russia. Born and raised in a small town in the city of Baltimore, Maryland, Franklin grew up with dreams to one day become a member of the

United States Department of Defense. After completing his Bachelor's degree in Military Science at the University of Maryland, he joined the Military. Next week marks his 7th consecutive year enlisted as an active member.

Due to high tensions between the Russian Regime and the United States, Franklin has been assigned the role of Command officer of a military unit about the size of 8 soldiers. He's tasked with scoping and mapping out a picture of how the battleground is constructed; As well as the detection of bomb-like structures within the battlefield before giving clearance to his team. He believes this software containing these capabilities, can be of immense use to not only his unit but other special operations in the military.

6. Appendix B – Copyright

Dr. rer. nat. Bastian Tenbergen,
Associate Professor of Software Engineering
bastian@tenbergen.org

Mikayla Conner-Spagnola, MA
Independent Consultant
mconner@oswego.edu

Department of Computer Science
State University of New York at Oswego
Oswego, NY 13126, United States