KomNAV HTTP API

Release 3.2

Kompai Robotics

May 10, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

This document details the HTTP API exposed by the Kompai robot navigation subsystem.

It is intended for developers that want to use the Kompai robots navigation functionalities through an HTTP interface.

This HTTP API is designed as possible with a RESTful approach in mind.

This translates to a specific way to use the HTTP requests.

The requests verbs (GET, PUT, POST, DELETE) are used to indicate some type of actions on the available resources.

- GET is used to READ content without modifying the state of the robot (e.g. reading the actual pose of the robot).
- POST is used to CREATE some content (e.g. adding a point of interest).
- PUT is used to UPDATE a content (e.g. specifying a new destination).
- DELETE is used to DELETE some content (e.g. removing a point of interest).

All content is formatted with JSON syntax, except for the map image.

Error handling is made through HTTP Status Codes.

This means that prior to using a response content, a client must check the HTTP response status code to be safe.

For periodic commands and measurements, the API also provide an access through WebSockets, typically available via a GET request on URLs that end with /socket.

CHAPTER

TWO

SENSORS AND MONITORING

2.1 Battery State

2.1.1 GET /api/battery/state

This request returns the actual battery state.

Response Content

Here is an example response:

```
{
    "Timestamp":0,
    "Status":3,
    "Remaining":56,
    "Autonomy":100,
    "Voltage":24.5,
    "Current":-1.2
}
```

- Status: the status of the battery.
  - 0 : Charging: the robot is being charged
  - 1 : Charged: The robot is fully charged but still connected
  - 2 : Ok: the battery is operating normally
  - 3 : Critical: the robot is about to shutoff
- Remaining: the percentage of energy remaining in the battery, range 0 - 100 %.
- Autonomy: the estimated remaining operation time, in minutes.
- Voltage: the actual battery voltage in Volts.
- Current: the actual current drawn from the battery, in Amps.

Status Code

200 - OK

2.1.2 GET /api/battery/socket

This requests allows negotiating a web socket to get notified each time a new battery state is available.

The format of the message is the same as for the GET /battery/state request.

2.2 Anticollision State

2.2.1 GET /api/anticollision/state

Response Content


```
{
    "Timestamp":1690040,
    "Enabled":true,
    "Locked":false,
    "Forward":0,
    "Reverse":0,
    "Left":0,
    "Right":0
}
```


- Enabled : Indicates if the low level anticollision is active (true) or not (false)
- Locked : Indicates if the robot desired motion is completely blocked (true) or not (false)
- Forward : Indicates if there is a a complete stop or partial limitation on forward motion, or no limitation:
  - 0 : There is no motion limitation
  - 1 : The speed is being limited, but motion is still allowed
  - 2 : Forward motion is completely forbidden
- Reverse, Left, Right: the same as Forward, but for reverse, left and right motions

Status Code

200 - OK

2.2.2 GET /api/anticollision/socket

This requests allows negotiating a web socket to get notified each time a new anticollision state is available.

The format of the message is the same as for the GET /api/anticollision/state request.

2.3 Laser Sensor

These requests allow retreiving the 2D LIDAR sensors measurements.

There are three sensors: The bottom laser merged with the 3D camera data, the top laser, and the bottom laser alone.

They can be accessed by changing {laser_name} in the requests URL:

- Bottom laser with 3D camera : laser
- Top laser : laser_top
- Bottom lase alone: laser_rear

2.3.1 GET /api/{laser_name}/properties

This request returns the properties of the laser sensor.

Response Content


```
{
    "Pose":
    {
        "X":-0.13199999928474426,
        "Y":0.0,
        "T":0.0
    },
    "MaxRange":10.0,
    "MinRange":0.01,
    "NumberOfEchoes":360,
    "MinAngle":-3.1241393,
    "MaxAngle":3.1415927
}
```


- Pose: Pose of the laser sensor in the robot frame.
- MaxRange: The maximum distance that can be reliably measured by the sensor
- MinRange: THe minimum distance that can be measured
- NumberOfEchoes: The number of measurements in on laser scan
- MaxAngle: Maximum angle of the mesurements
- MinAngle: Minimum angle of the mesurements

The robot frame X axis is directed in front of the robot, the Y axis on the left.

2.3.2 GET /api/{laser_name}/state

Response Content


```
{
    "Timestamp":1103160,
    "Echoes":
    [
        {"Distance":2.2800000000000002,"Angle":-1.5707963705062866},
        {"Distance":2.2800000000000002,"Angle":-1.5649785995483398},
        {"Distance":2.2403571143904712,"Angle":-1.5591608285903931}
    ]
}
```


- Timestamp: this the laser scan timestamp in milliseconds since robot startup.
- Echoes: this is an array of angle and distance mesaurements, in the sensor frame. Distance is in meters and Angle is in radians.

Status Code

200 - OK

2.3.3 GET /api/{laser_name}/socket

This requests allows negotiating a web socket to get notified each time a new laser scan is available.

The format of the message is the same as for the GET /api/laser/state request.

2.4 I/O devices

2.4.1 GET /api/io/properties

This request returns the names of the robot I/O s.

Response Content


```
{
    "DIn":
    [
        "DIn_0",
        "DIn_1",
        "DIn_2",
        "DIn_3",
        "DIn_4",
        "DIn_5",
        "DIn_6",
        "DIn_7",
        "A",
```


(continues on next page)

(continued from previous page)


```
        "B",
        "X",
        "Y"
    ],
    "DOut":[],"AIn":["AIn"],"AOut":[]
}
```


The DIn_x inputs are logicla inputs of the motor drive.

DIn_2 is the stop button on the right robot handle.

The A, B, X and Y inputs are the state of the gamepad buttons.

2.4.2 GET /api/io/state

This returns the I/O values.


```
{
    "Timestamp":375950,
    "DIn":
        [true,true,true,false,false,true,true,true,false,false,false,false],
    "AIn":[0.0]}
```


- Timestamp: time of the data in milliseconds
- DIn: logical inputs states
- AIn: analog inputs values

2.4.3 GET /api/io/socket

This opens a websocket to get the I/O state, in the same format as the GET /api/io/state request.

2.5 Operation statistics

These requests return the robot total operating time and distance covered since its commissioning.

2.5.1 GET /api/statistics/state

Response Content


```
{
    "Timestamp":2504150,
    "TotalTime":247832,
    "TotalDistance":2181
}
```


- Timestamp: Time in milliseconds since robot startup
- TotalTime: Total operation time in seconds

- TotalDistance: Total distance covered, in meters

2.5.2 GET /api/statistics/socket

Opens a websocket to get statistics data.

2.6 Data logging

2.6.1 GET /api/logs?start={date time}&duration={seconds}

This request returns a zip file with process data logs.

CHAPTER

THREE

LASER BASED LOCALIZATION

This chapter describes the laser based localization subsystem.

This subsystem allows localizing the robot in a 2D metric map describing obstacles, by comparing it to the laser sensor data.

In order to use this subsystem, you need:

- An available map of the environment
- To upload it to komNAV throught POST /api/maps/list
- You must then load the map with PUT /api/maps/current/id or PUT /api/maps/current/name
- Once loaded, you need to initialize the robot pose with PUT /api/localization/pose.

3.1 Maps Management

3.1.1 GET /api/maps/list

This request returns a list of maps and their properties.

Response Content

The response contains an array of maps properties.


```
[
    {
        "Id":2,
        "Resolution":0.04,
        "Name":"Izarbel",
        "Width":893,
        "Height":1296,
        "Offset":
        {
            "X":-20.0,
            "Y":-20.0
        }
    },
    {
        "Id":31,
        "Resolution":0.05,
```


(continues on next page)

(continued from previous page)


```
        "Name":"Office",
        "Width":277,
        "Height":360,
        "Offset":
        {
            "X":0.0,
            "Y":0.0
        }
    }
]
```


A map is described by this parameters:

- Id : a unique identifier, created when the map is uploaded
- Resolution : the size of the area represented by a single pixel, in meters. For example 0.04 means 4 centimeters
- Name : A unique name
- Width : the width of the map image in pixels
- Height : the height of the map image in pixels
- Offset : The position of the lower left corner of the map in the map frame, in meters.

Response Codes

200 - Ok

3.1.2 POST /api/maps/list

This allows uploading a map.

Request Content

This URL expects a multipart form data.

The expected parts are:

- name : A unique character string naming the map
- resolution : the size of a pixel, in meters
- offset_x : the x position of the lower left corner of the map, in meters
- offset_y : the y position of the lower left corner, in meters
- filename : the map image file

Response Content

The response body contains the Id of the newly uploaded map.

Response Codes

201 - Created: the map was successfully saved. 400 - Bad Request: the parts couldn't be read or a map with this name already exists.

3.1.3 PUT /api/maps/list/{id}

Request Content

The content is the same as the POST /api/maps/list request.

3.1.4 DELETE /api/maps/list/{id}

This request allows deleting an existing map.

The request body is empty, the {id} part of the URL should be replaced with the Id field of the map that must be deleted.

Response Codes

200 - Ok : the map has been deleted 400 - Bad Request : the map doesn't exist

3.1.5 PUT /api/maps/current/name/{name}

This request allows selecting a map to initialize the navigation subsystem.

The {name} part of the URL must be replaced by the name of an existing map.

You can add the query parameter autoinit=false if you don't want komNAV to try to initialize the localization.

3.1.6 PUT /api/maps/current/id/{id}

Same as the previous request, but with a map id instead of name.

3.1.7 GET /api/maps/current/properties

Returns the current map properties, in the same format used for GET /api/maps/list.

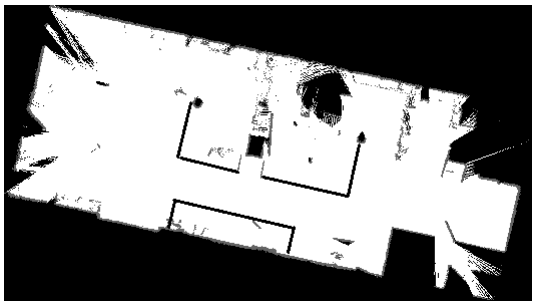
Response Codes

200 - OK

404 - Not Found : the robot hasn't loaded a map.

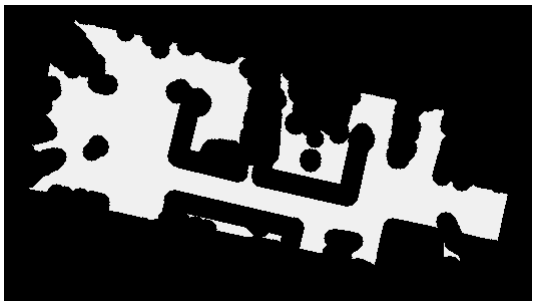
3.1.8 GET /api/maps/current/image

This request returns the map image in PNG format.



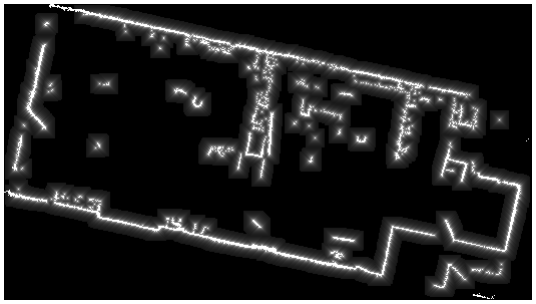
3.1.9 GET /api/maps/current/planner/image

This request returns the map used by the planner image in PNG format. It can be useful to check if there are places too narrow for the robot to pass.



3.1.10 GET /api/maps/current/localizer/image

This request returns the map used by the localizer image in PNG format.



3.1.11 GET /api/maps/current/locations

This request returns the list of Locations of interest associated with current map.

Response Content

The content is an array of locations description records.


```
[
    {
        "Id":24,
        "Map":0,
        "Name":"Office",
        "Label":"",
        "Pose":
        {
            "X":1.75,
            "Y":2.35,
            "T":0.0
        }
    },
    {
        "Id":25,
        "Map":0,
        "Name":"Boites",
        "Label":"",
        "Pose":
        {
            "X":8.6,
            "Y":17.5,
            "T":0.0
        }
    }
]
```


For each record, the fields are:

- Id : a unique identifier created when the location is added
- Map : the identifier of associated map
- Name : a string identifier for the location
- Label : an arbitrary string
- Pose : the pose of the location in the map

Status Code

200 - Ok

3.1.12 POST /api/maps/current/locations

This request allows adding a location.

Request Content

```
{
    "Name" : "bathroom",
    "Pose" :
    {
        "X" : 1.2,
        "Y" : 3.4,
        "T" : 0.1
     },
     "Label" : "..."
}
```

Response Content

This returns the id generated for the location.

Response Codes

200 - Ok

400 - Bad Request : a point of interest with the same name alreay exists

3.1.13 PUT /api/maps/current/locations/{id}

This request allows updating and existing location.

The content is the same as the POST /api/maps/current/locations request.

3.1.14 DELETE /api/maps/current/locations/{id}

This request allows removing alocation. The {id} part of the URL should be replaced with Id field of the location to delete.

Response Codes

200 - OK

400 - Bad Request : Couldn't decode the id.

3.2 Laser Localization

3.2.1 GET /api/localization/state

This request returns the actual localization of the robot.

Response Content

```
{
    "X" : 1.2,
    "Y" : 3.4,
    "T" : 0.5,
    "Localized" : true
}
```

- X : X coordinate in meters
- Y : Y coordinate in meters
- T : Orientation in radians
- Localized : true if the pose is corrected with laser localization, false if only odometry

Status Code

200 - OK

3.2.2 PUT /api/localization/pose

This request resets the robot localization with the specified pose.

Request Content

```
{
    "X" : 1.2,
    "Y" : 3.4,
    "T" : 0.5
}
```

Status Code

202 - Accepted

3.2.3 GET /api/localization/socket

Allows negotiating a web socket to be notified each time a new pose is available.

CHAPTER

FOUR

MOTION COMMANDS

The robot can be controlled using two modes, either with direct speed commands, to use with a gamepad for example, or through position targets, where the robot plans and execute a trajectory to reach the desired position.

There is an extra mode for automatic docking to the charging station.

4.1 Direct speed control

This allows interacting with the robot in direct speed control mode. The robot is controlled by specifying a linear speed and an angular speed.

4.1.1 GET /api/differential/properties

This returns the robot speed control properties.

Response Content


```
{
    "MaxLinearSpeed":1.0,
    "MinLinearSpeed":-1.0,
    "MaxAngularSpeed":1.5,
    "MinAngularSpeed":-1.5,
    "MaxLinearAcceleration":0.75,
    "MaxLinearDeceleration":-0.75,
    "MaxAngularAcceleration":3.0,
    "MaxAngularDeceleration":-3.0,
    "Width":0.4239675998687744
}
```


- MaxLinearSpeed: maximum speed in m/s
- MinLinearSpeed: minimum speed in m/s
- MaxAngularSpeed: maximum angular speed in rad/s
- MinAngularSpeed: minimum angular speed in rad/s
- MaxLinearAcceleration: maximum linear acceleration in m/s^2
- MaxLinearDeceleration: maximum linear deceleration in m/s^2
- MaxAngularAcceleration: maximum angular acceleration in rad/s^2

- MaxAngularDeceleration: maximum angular deceleration in rad/s^2
- Width: width of the robot on meters

4.1.2 PUT /api/differential/command

This request sets the immediate speed target of the robot. To avoid buffering effects, resulting in an erratic behaviour of the robot, it is recommended to always wait for a response to this request before sending a new one.

The PUT /navigation/stop request has no effect on this service.

If too much time passes between two PUT /differential/command, the robot will stop and disable the motors by himself.

Request Content

```
{
    "Enable":true,
    "TargetLinearSpeed":0,
    "TargetAngularSpeed":0.2
}
```

- Enable: if set to true, the robot will enable the motor drives and disengage the brakes. If false, the robot will shutdown the motor drives and stop providing them energy, after a 1 second delay. If set to false, the other parameters have no effect. Note that if the robot is moving when transitionning from Enable = true to false, the robot will first set its target speeds to zero and decelerate before engaging the brakes.
- TargetLinearSpeed: the desired forward (or reverse, if negative) speed in meters/second. Note the robot will apply its configured ramps to the specified target.
- TargetAngularSpeed: the desired rotation speed in radians/second, in the trigonometric direction.

4.1.3 GET /api/differential/state

This request returns the current state of the differential drive controller.

Response Content

```
{
    "Timestamp":1984190,
    "Status":1,
    "TargetLinearSpeed":0.5,
    "CurrentLinearSpeed":0.5,
    "TargetAngularSpeed":-0.041446387767791748,
    "CurrentAngularSpeed":-0.041446343064308167
}
```

- **Status: it can be three different values:**
  - 0 : Disabled : the motors are disabled
  - 1 : Enabled : the motors are enabled and executing the desired targets

- 2 : Error : something wrong happened, so the motors are stopping if possible. It is possible to exit this state by transitionning the "Enable" field in the command from true to false. When the error condition disappear it then possible to switch to Enabled again.

Status Code

200 - Ok

4.1.4 GET /api/differential/socket

This allows negotiating a websocket that can both receive state updates and send commands, with the same format as the previous GET /api/differential/state and PUT /api/differential/command requests.

4.2 Step Control

Step control allows the robot to make a single rotation or translation.

4.2.1 GET /api/step/state

This returns the state of the step controller.

Response Content

```
{
    "Timestamp":55580,
    "CommandTimestamp":5285,
    "Travelled":0.25260013341903687,
    "Motion":1,
    "Status":0
}
```

- Timestamp: time of the state
- CommandTimestamp: tick at which the last command was received
- Travelled: Distance or Angle that has been covered
- Motion: Type of motion; 0 - Translation, 1 - Rotatation
- State: 0 - Currently moving, 1 - Waiting, 2 - Error

4.2.2 PUT /api/step/translate

This request orders a translation.

Request Content

```
{
    "Distance" : 1.5,
    "Speed" : 0.3
}
```

- Distance: The distance to travel in meters
- Speed: The maximum speed in meters/second

4.2.3 PUT /api/step/rotate

This request orders a rotation.

Request Content

```
{
    "Angle" : 0.75,
    "Speed" : 0.2
}
```

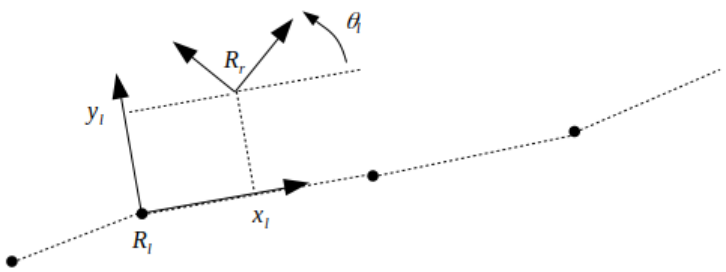
- Angle: The angle to travel in radians
- Speed: The maximum speed in radians/second

4.2.4 PUT /api/step/stop

This request stops any current step motion.

4.3 Trajectory Following

The trajectory following algorithm controls the robot to follow a series of points specified by the user.



4.3.1 GET /api/trajectory/state

This request returns the state of the trajectory following algorithm.

Response Content


```
{
    "Timestamp":507180,
    "ControlTimestamp":7225,
    "Control":2,
    "PathTimestamp":7240,
    "Status":1,
    "DistanceCovered":12.021647141718974,
    "Error":
    {
        "X":0.0,
        "Y":0.0,
        "T":0.0},
    "ErrorFlags":0
}
```


- Timestamp : Time of the state
- ControlTimestamp : Time at which the last control was received
- Control : Current control value (see below for details)
- PathTimestamp : Time at which a path was last received

4.3.2 PUT /api/trajectory/follow

This request allows sending a path to follow.

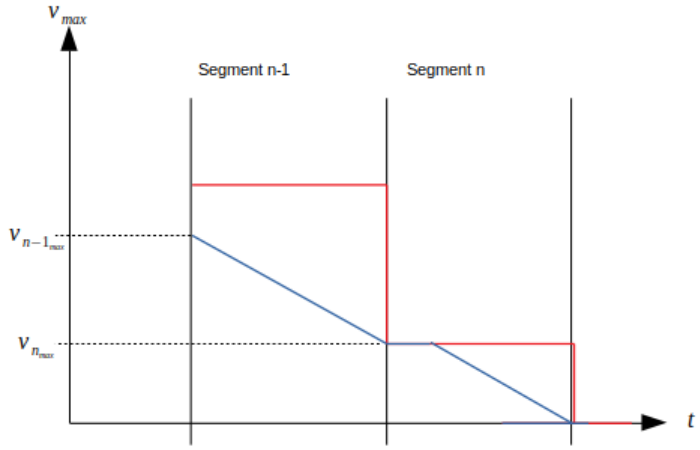
Request Content

The request contains an array of segments, defined by a start point, an end point and a maximum speed.


```
[
    {
        "Start":
        {
            "X":6.6,
            "Y":16
        },
        "End":
        {
            "X":15,
            "Y":7.4
        },
        "MaxSpeed":1
    }
]
```

- Start : The strat point of the segments
- End : the end point of the segments
- MaxSpeed : the maximum speed, in meters/second

The algorithm will ensure that the maximum speed is never exceeded, as shown in he figure below.



4.4 Autonomous Motion

4.4.1 GET /api/navigation/state

This request returns the state of a motion to a destination.

Response Content


```
{
    "Status":1,
    "Avoided":0,
    "Trajectory":
    [
        {
            "Start":
            {
                "X":3.3399999999999976,
                "Y":2.8600000000000008
            },
            "End":
            {
                "X":5.7399999999999993,
                "Y":4.1499999999999977
            },
            "MaxSpeed":0.5
        },
        {
```


(continues on next page)

(continued from previous page)


```
            "Start":
            {
                "X":5.7399999999999993,
                "Y":4.1499999999999977
            },
            "End":
            {
                "X":6.5799999999999992,
                "Y":4.4499999999999993
            },
            "MaxSpeed":0.5
        }
    ]
}
```


The status can have the following values:

- 0 - Waiting: the robot is ready for a new destination specified with a PUT /navigation/destination request.
- 1 - Following: the robot is following the trajectory computed to join the destination.
- 2 - Aiming: the robot has completed the trajectory and is now rotating to aim the destination accurately
- 3 - Translating: the robot has finished aiming and is now translating to reach the destination
- 4 - Rotating: this is the last rotation, executed only if the destination was specified with "Rotate" : true
- 5 - Error: indicates that the robot has encountered an error that prevented it to join the required destination. This can be cleared by sending a new destination.

The "Avoided" field indicates how many times a trajectory was recomputed before reaching destination to avoid an obstacle.

Each time this field is incremented, there is an updated trajectory available at /navigation/trajectory. After succesfully planning a trajectory upon a PUT on /navigation/destination, the counter is reset to zero. After the first obstacle avoided it will be set to 1, etc.

The "Trajectory" field contains the trajectory computed by the path planner.

Status Code

200 - Ok

4.4.2 GET /api/navigation/socket

This allows negotiating a web socket to be notified each time a new navigation state is available, in the same format as the GET /api/navigation/state request.

4.4.3 PUT /api/navigation/destination/pose

This request allows specifying a pose destination for the robot.

Request Content

The request contains the desired destination.

```
{
    "X" : 1.2,
    "Y" : 3.4,
    "T" : 0.5
}
```

- X, Y, T: the coordinates of the destination

Status Code

200 - Ok 400 - BadRequest: the robot can't execute the request

4.4.4 PUT /api/navigation/destination/point

It's the same as the previous request, but it won't try to reach the requested orientation, only the position.

4.4.5 PUT /api/navigation/destination/name/{name}

The robot will try to join the location indicated by {name}.

Status Code

200 - Ok 400 - BadRequest: the robot can't execute the request 404 - The location is unknown 500 - There was an error executing the request

4.4.6 PUT /api/navigation/abort

This request allows aborting motion to a destination set with the PUT /api/navigation/destination/pose request.

Status Code

200 - Ok

4.4.7 PUT /api/navigation/dock

This will send the robot to its docking station and initiate the docking procedure. Note that thios will work only if the current map has a "docking" location defined.

4.5 Charging Station Docking

This mode allows the robot to connect autonomously to its charging station, provided it is in front of it.

4.5.1 GET api/docking/state

Response Content

This contains the state of the docking controller.


```
{
    "Timestamp":17888300,
    "Status":1,
    "Pose":null,
    "Detected":false
}
```


- Timestamp: time of the state, in milliseconds
- **Status:**
  - 0 - Unknown: Status is unknown
  - 1 - Undocked: The robot is not docked
  - 2 - Docking: The robot is currently docking
  - 3 - Docked: The robot is docked
  - 4 - Undocking: The robot is leaving the docking
  - 5 - Error: There was an error while attempting to dock

4.5.2 GET api/docking/socket

This allows opening a Websocket to get the state of the docking controller.

The data has the same format as the GET /api/docking/state response.

4.5.3 PUT /api/docking/connect

This starts the docking process.

4.5.4 PUT /api/docking/disconnect

This instructs the robot to disconnect from the docking.

4.5.5 PUT /api/docking/abort

This aborts the docking or undocking process, if one is currently executing.

4.6 Walking Assistance

This section describes the walking assistance modes.

The first mode lets the user freely move to any direction, adjusting the robot speed and heading with respect to the user movements.

The second mode adjusts only the speed based on the user movements, and adjust the heading to guide the user to a destination, using the autonomous motion controller.

4.6.1 GET /api/walker/state

Response Content

This contains the state of the walking assistance controller.


```
{
    "Timestamp":0,
    "Status":0,
    "Origin":
    {
        "X":0.0,
        "Y":0.0
    },
    "Person":
    {
        "X":0.0,
        "Y":0.0
    },
    "Detected":false
}
```


- Timestamp: time of the state, in milliseconds
- **Status:**
  - 0 - Disabled: the walking assitance controller is disabled
  - 1 - Enabled: the controller is enabled, but not issuing motion commands
  - 2 - Error: there was an error during assistance
  - 3 - Active: the assistance is active

4.6.2 GET /api/walker/socket

This opens a websocket to get the state of the walking assistance controller.

The data has the same format as the GET /api/walker/state response.

4.6.3 PUT /api/walker/command

This request allows enabling and activating the walking assistance.

Request Content

```
{
    "Enable": true,
    "Guided": false,
    "Assist": true,
    "MaxSpeed" : 0.75
}
```

- Enable: Enables the controller when true. Thee controller starts detecting persons but does not yet issue speed commands.
- Guided: if false, the walking assistance controller will regulate both the rotation and the linear speed of the robot.

  If true, it will regulate only the linear speed, and let the rotation speed be controlled by the navigation controller (See Autonomous Motion section).

  This allows guiding a person to a deired location.
- Assist: when set to true, the controller will start issuing commands. This is only possible if the field detected in the controller state is set to true.
- MaxSpeed: Maximum speed percentage, must be between 0 and 1.

4.6.4 PUT /api/exercise/start/{id}

This request allows following a path specified by the locations in the round specified with {id}.

It will not use the path planner, but instead generate some trajectories between the locations, and then chain them without stopping until the last one.

4.6.5 PUT /api/exercise/pause

This request allows pausing the exercise. The robot will stop but thexercise can be resumed with PUT /api/exercise/resume request.

4.6.6 PUT /api/exercise/resume

Thois request allows resuming an exercise.

4.6.7 GET /api/exercise/state

This request allows getting the state an exercise.

Notification Content


```
{
    "Timestamp": 12000,
    "Status" : 1,
    "Trajectory":
    [
        {
            "Start":
            {
                "X":3.3399999999999976,
                "Y":2.8600000000000008
            },
            "End":
            {
                "X":5.7399999999999993,
                "Y":4.1499999999999977
            },
            "MaxSpeed":0.5
        },
        {
            "Start":
            {
                "X":5.7399999999999993,
                "Y":4.1499999999999977
            },
            "End":
            {
                "X":6.5799999999999992,
                "Y":4.4499999999999993
            },
            "MaxSpeed":0.5
        }
    ]
}
```


The different fields are:

- Timestamp: Time at which the state was generated
- Status: 0 - Waiting, 1 - Following, 2 - Paused, 3 - Error
- Trajectory: The complete trajectory that is generated

4.6.8 GET /api/exercise/socket

This request allows getting a socket to get notifications about the state of an exercise. The content is the same as the GET /api/exercise/state request.

CHAPTER

FIVE

ROUNDS CONFIGURATION AND EXECUTION

This chapter describes the so called “Rounds” low level functionality.

The idea of a round is to instruct the robot to navigate through a series of Points of Interests, and have it perform some actions at each one.

A client application can configure a round by specifying PoI that must be reached, along some data on actions to be performed.

Currently the data supported are a speech text, a speech volume and media volume.

5.1 Adding a Round

5.1.1 POST /api/rounds/list

This request adds a round in the database.

Request Content

```
{
    "Name":"test2",
    "Map":8,
    "Locations":
    [
        {
            "Speed":0.01,
            "Wait":1,
            "Data":"test",
            "DisableAvoidance":true,
            "AnticollisionProfile": 0,
            "Location":{"Id":24}
        },
        {
            "Speed":1,
            "Wait":1,
            "Data":"test",
            "DisableAvoidance":true,
            "AnticollisionProfile": 1,
            "Location":{"Id":28}
        }
```

(continues on next page)

(continued from previous page)

```
    ]
}
```

The fields are the following:

- Name : A name that can be used for display purpose
- Map : The id of the map used for the round
- Locations : An array of locations to visit.

A locations can be used several times.

A location record contains the following parameters:

- Speed : a percentage of the maximum speed
- Wait : the duration to wait after reaching the location, in seconds
- Data : at text string that the client wants to store
- DisableAvoidance : disables the obstacle avoidance algorithm while reaching the location
- AnticollisionProfile : an integer that gives the anticollision profile index (0 for default, 1 for reduced)
- Location : the location to reach

5.1.2 PUT /api/rounds/list/{id}

This request will update an existing round.

5.1.3 DELETE /api/rounds/list/{id}

This request will delete an existing round.

5.1.4 GET /api/rounds/list

This request returns the list of rounds stored in the database.

Response Content

```
[
    {
        "Id":1,
        "Map":1,
        "Name":"test1",
        "Locations":[
        {
            "Speed":0.01,
            "Wait":1,
            "Data":"test",
            "DisableAvoidance":true,
            "AnticollisionProfile": 0,
            "Location":{"Id":24}
```

(continues on next page)

(continued from previous page)

```
        },
        {
            "Speed":1,
            "Wait":1,
            "Data":"test",
            "DisableAvoidance":true,
            "AnticollisionProfile": 1,
            "Location":{"Id":28}
        }]
    }
]
```

It contains an array of rounds.

5.2 Executing A Round

5.2.1 PUT /api/rounds/current/id/{id}

This request will start the round with the identifier {id}.

Response Codes

200 - Ok 400 - Bad Request: the id doesn't exist

5.2.2 PUT /api/rounds/current/acknowledge

This will instruct the robot to move to the next PoI, when waiting on the current one. The client must call this when the round status is OnLocation, and the client has completed its tasks.

5.2.3 PUT /api/rounds/current/pause

This will pause the round, and its status will change to Paused. This can be called when the round status is either Moving or OnLocation.

5.2.4 PUT /api/rounds/current/resume

This will resume the round after a call to pause. The robot will restart moving toward the last PoI that hasn't been acknowledged. The round status must be Paused to use this call.

5.2.5 PUT /api/rounds/current/abort

This call will abort the round. To start a new one, the client must use th POST /api/rounds/current/id/ call.

5.2.6 PUT /api/rounds/current/skip

This request will skip the current location that the robot is trying to reach and reroute to the next one.

5.2.7 GET /api/rounds/state

This request retuns the round state.

Request Content


```
{
    "Round":
    {
        "Id":2,
        "Map":1,
        "Name":"test2",
        "Locations":
        [
            {
                "Speed":1,
                "Wait":1,
                "Data":"test",
                "DisableAvoidance":true,
                "Location":{"Id":28}
            }
        ]
    },
    "Acknowledge":false,
    "Abort":false,
    "Pause":false,
    "Status":2
}
```


- Round : this is the entry from GET /api/rounds/list/ that has been selected with PUT /api/rounds/current/id
- Acknowledge: indicates if the location has been acknowledged
- Abort : indicates if the round has been aborted
- Pause : indicates if the round is paused
- **Status:** 0 - None: there is no round set 1 - Ready: round is ready to be executed 2 - Moving: the robot is moving to the next PoI 3 - Paused: the round has been paused 4 - OnLocation: the robot is on the PoI waiting an acknowledged 5 - Completed: the round has been completed
- Locations : the remaining locations to visit

5.2.8 GET /api/rounds/socket

This request opens a websocket to get notified of the round state.

CHAPTER

SIX

MISCELLANEAOUS FUNCTIONS

6.1 System testing

There is a builtin self test where the robot will automatically reach all locations in a loaded map repeatedly, and go to its docking when battery is low.

6.1.1 PUT /api/systemtest/run

This will start the system test.

6.1.2 PUT /api/systemtest/stop

This will stop the system test.

6.2 Eyes control

6.2.1 PUT /api/eyes?id={id}

This request allows setting the image displayed by the Kompai eye.

6.2.2 DELETE /api/eyes?id={id}

This request allows removing he image displayed by the Kompai eye.

6.2.3 List of possible eyes

The {id} field must be one in the following list:

- 0 - WaitListen
- 1 - Alarm 1
- 2 - MotionComplete
- 3 - BatteryCharging
- 4 - BatteryLow
- 5 - BatteryCritical

- 6 - BatteryFull
- 7 - BatteryEmpty
- 8 - DrugConfirm
- 9 - MotionInProgress
- 10 - InternetUnavailable
- 11 - Error
- 12 - SpeechInterrogation
- 13 - DrugTake
- 14 - ObstacleBlocking
- 15 - SpeechGeneral
- 16 - RemoteControl
- 17 - LocalizationLostLeft
- 18 - LocalizationLostLeft2
- 19 - CommunicationTerminated
- 20 - SafetyCheck
- 21 - SOS
- 22 - NoIcon
- 23 - CommunicationInProgress
- 24 - CommunicationIncoming
- 25 - ScreenAttention

Several icons can be set at the same time, but only the one with the lowest identifier will be displayed.

CHAPTER

SEVEN

INSTALLING AND RUNNING KOMNAV SERVER

7.1 Installing .NET Core and komNAV Server

komNAV Server is delivered as a .zip archive containing the application. You can unzip to any convenient location.

In order to run, it needs the .NET Core runtime version 2.2.2.

7.1.1 On Windows

You can download the runtime installer here:

https://dotnet.microsoft.com/download/thank-you/dotnet-runtime-2.2.2-windows-x64-asp.net-core-runtime-installer

You also need to authorize komNAV Server to receive HTTP request on a port (7007 is the default):

In a command prompt with administrative privilege, enter this command:

```
netsh http add urlacl url=http://+:7007/ user=PC-MARC\marc
```

Warning: Replace PC-MARCmarc with the user account that will run komNAV Server

7.1.2 On Ubuntu Linux

You need to add Microsoft packages repository, and install the packages:

```
wget -q https://packages.microsoft.com/config/ubuntu/16.04/packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
sudo apt-get install apt-transport-https
sudo apt-get update
sudo apt-get install aspnetcore-runtime-2.2
```

7.2 Running komNAV Server

komNAV Server can be configured through a JSON file, with the following format:

```
{
    "LogLevel": 0,
    "PureAdress": "127.0.0.1",
    "PurePort": 60000,
    "ConfigurationDatabaseFilename": "/home/nvidia/komnav/komnav.sqlite3",
    "LogsDatabaseFilename": "/home/nvidia/komnav/komnav.logs.sqlite3",
    "NavigationMap": "atelier",
    "WebServerPort": 7007,
    "WebServerRoot": "../dashboard",
    "SimulateLaser": false,
    "RobotDiameter": 0.75,
    "BatteryLowLevel": 40,
    "BatteryHighLevel": 80
}
```

- **LogLevel: Verbosity of the console logs:**
  - 0 - Verbose : all messages are logged
  - 1 - Info : information messages are logged
  - 2 - Warning : information that could indicate abnormal operation
  - 3 - Error : unexpected errors that shouldn't happen
  - 4 - Critical : events that cause the application to shut down
- PureAdress: IP address of the low level controller
- PurePort: UDP port used by the low level controller
- ConfigurationDatabaseFilename: file containing the configuration database (maps, locations, rounds)
- LogsDatabaseFilename: file containing the logging database
- NavigationMap: an eventual navigation map to load at startup, set to null if unwanted
- WebServerPort: the port on which to listen for HTTP requests
- WebServerRoot: The directory from which files will be served
- SimulateLaser: The laser sensor data will be generated frome the map, useful when using a basic simulator with no laser data.
- RobotDiamater: The diameter of the robot to use for the path planning and obstacle avoidance
- BatteryLowLevel: Battery level to go docking in system test
- BatteryHighLevel: Battery level to leave docking in system test

If ran without arguments, komNAV Server will search for a file named configuration.json in its binaries directory. A default one should be provided in the zip archive. If it's not found it will create a default configuration.json in its directory and use it.

Once configured, komNAV Server can be run with the following command:

```
dotnet komnav_server.dll
```

CHAPTER

EIGHT

USING THE KOMNAV SERVER WEB DASHBOARD

8.1 Basic Usage

On top of the HTTP API described in this document, komNAV Server provides a simple dashboard written in HTML, Javascript and CSS that is available through a standard Web Browser, like Mozilla Firefox or Google Chrome.

To access this dashboard, open a browser at the following URL: http://localhost:7007/

This is supposing that komNAV server is running on the same machine as the browser and that the default HTTP port hasn't been changed.

In that case you should get the following page:

Fig. 1: Dashboard landing page

The following sections describe each tab.

8.2 Diagnostics Tab

This tab gives some summary of the robot status, like its battery state and devices status.

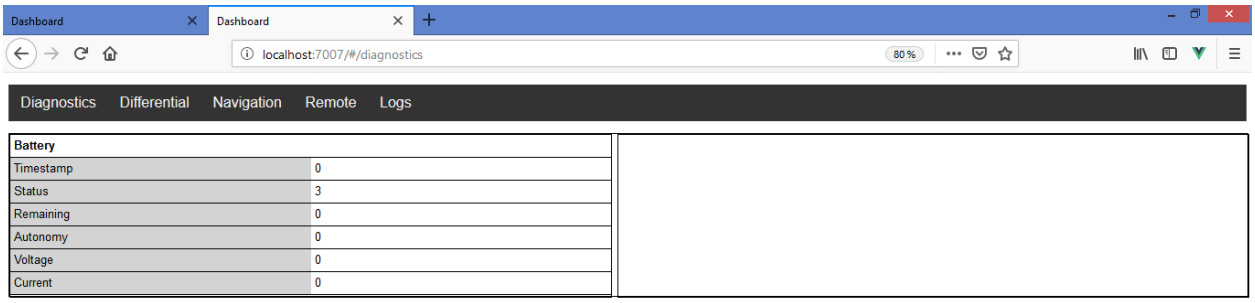


Fig. 2: Diagnostics Tab

8.3 Differential Page

This tab gives some details on the differential control of the robot.

The buttons on the upper left allows controlling the robot speed by maintaining them pressed.

The Abort button will stop any automous motion.

The lower left grid shows the current state of the differential controller.

The upper center graph shows the target linear speed (blue line) and the actual linear speed (red line).

The lower center graph shows the target angular speed (blue line) and the actual angular speed (red line).

The upper right grid gives the properties of the differential controller. These are statically configured in the low level controller.

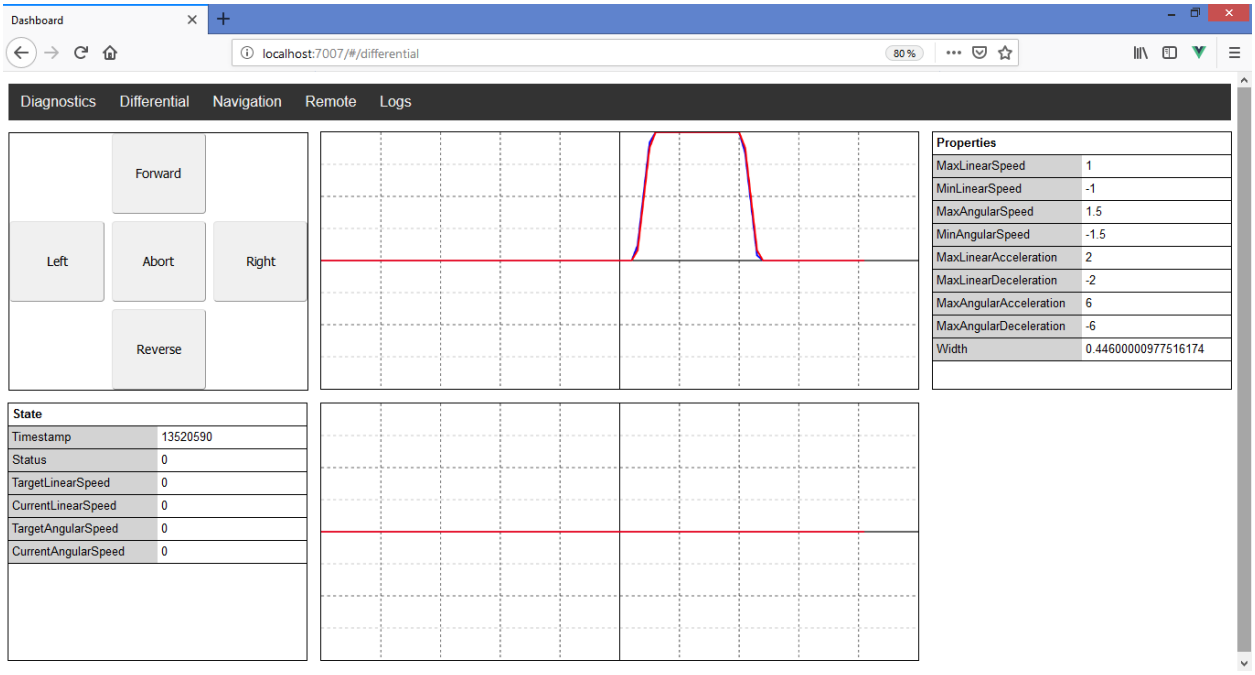


Fig. 3: Differential Tab

8.4 Navigation Page

This tab gives access to the autonomous navigation functions.

The right part is dedicated to maps and current map locations handling.

The upper left part is dedicated to operations on the current point.

The current point can be set by:

- left-clicking in the map, in which case the current point is set to the coordinates clicked
- clicking on the third button “Set Current As Robot”, in which case the current point is set the same as the actual robot pose

The other buttons functions are:

- “Reach Current Point” : ask the robot to plan a path and execute it to reach the current point
- “Add Current Point” : add the current point as a location in the current map. A form will pop up asking for a name to give to the location.
- “Reset Robot at Current Point” : resets the robot pose at the current point; this is necesary at start up and useful if the robots gets lost because of a changing environment for example.

The central area displays various information about the robot:

- The light blue shape gives the robot pose, it’s orientation is indicated by the triangle shaped side.
- The green point is the current point, selected by left clicking
- The red points are the laser sensor measurements
- The blue points are the current map locations

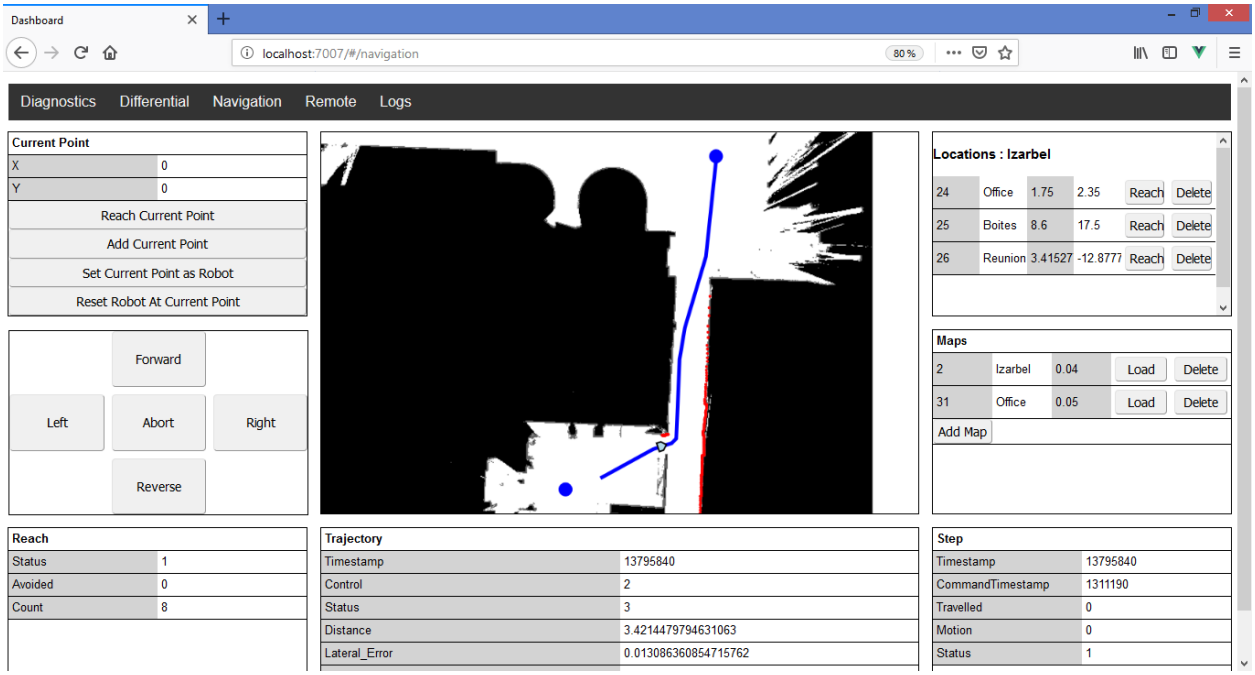


Fig. 4: Navigation Tab

- The blue line is the last trajectory executed by the robot.

It is possible to move the map by pressing on the mouse middle button and dragging in the desired direction.

It is possible to zoom or unzoom using the mouse wheel.

8.5 Remote Page

This page allows watching a video stream from the robot.

> **Warning:** In order to work, there must be another browser connected to http://localhost:7007/local/local.html

8.6 Some Information about the Dashboard Implementation

This dashboard can also be used as an example of how to use the HTTP API.

It is implemented using Vue.js for the graphical components, Vue-router.js for tab handling.

The entry point is the file index.js. Most of the API calls are grouped in it. They can be used as examples of the API usage.

The files are organized with the following directory structures:

- **dashboard:**
  - **external: external libraries**
    - vue.js

    * vue-router.js
    * jquery.js
- **components: vue.js graphical components**
    * diagnostics-view.js : devices status
    * differential-buttons.js : the button box to move the robot
    * map-list.js : the map list display, to add, load or delete a map in the navigation page
    * map-view.js : the map display in the center of the navigation page
    * oscilloscope.js : the graphing component in the Differential page
    * struct-display.js : used in all pages to display data structures
    * video_view.js : displays a video stream
- index.js: the main script that initializes the application
- Diagnostics.js : the root component of the Diagnostics page
- Differential.js : the root component of the Differential page
- Navigation.js : the root component of the Navigation page