**Документация NavBot WebSocket Client API**

**Обзор**

rf\_bot\_api.py предоставляет клиентский API для управления роботом через WebSocket соединение. Клиент имитирует интерфейс оригинального NavBot API, позволяя управлять роботом удалённо с той же простотой использования.

**Установка и подключение**

**Импорт и инициализация**

from nav\_api\_ws import NavBot  
  
# Подключение к локальному серверу  
bot = NavBot()  
  
# Подключение к удалённому серверу  
bot = NavBot(host="192.168.0.10", port=8765)

**Параметры конструктора**

* **host** (str, по умолчанию "localhost"): IP-адрес или доменное имя сервера
* **port** (int, по умолчанию 8765): Порт для WebSocket соединения

**Методы управления роботом**

**1. Инициализация и ожидание готовности**

**wait\_until\_ready(timeout=None)**

Ожидает готовности робота к работе.

**Параметры:**

* timeout (float | None): Максимальное время ожидания в секундах. При None - ожидание без ограничений

**Возвращает:** None

**Пример использования:**

bot.wait\_until\_ready() # Ожидание без ограничения времени  
bot.wait\_until\_ready(10.0) # Ожидание до 10 секунд

**2. Получение информации о положении**

**get\_pose()**

Получает текущее положение и ориентацию робота.

**Возвращает:**

* Tuple[float, float, float]: (x, y, yaw) - координаты и угол поворота

**Пример использования:**

x, y, yaw = bot.get\_pose()  
print(f"Робот находится в точке ({x:.2f}, {y:.2f}) с углом {yaw:.2f} рад")

**3. Проверка доступности цели**

**check\_target(x, y)**

Проверяет, доступна ли указанная точка для навигации и возвращает расстояние до неё.

**Параметры:**

* x (float): X-координата целевой точки
* y (float): Y-координата целевой точки

**Возвращает:**

* Tuple[bool, float]: (доступность, расстояние)

**Пример использования:**

available, distance = bot.check\_target(2.0, 1.5)  
if available:  
 print(f"Цель доступна, расстояние: {distance:.2f} м")  
else:  
 print(f"Цель недоступна, расстояние: {distance:.2f} м")

**4. Сканирование окружения**

**check\_around()**

Получает массив расстояний до препятствий вокруг робота.

**Возвращает:**

* List[float]: Список расстояний в разных направлениях

**Пример использования:**

distances = bot.check\_around()  
print(f"Расстояния до препятствий: {distances}")  
  
# Проверка минимального расстояния  
min\_distance = min(distances)  
if min\_distance < 0.5:  
 print("Внимание: препятствие очень близко!")

**5. Навигация**

**navigate(x, y, yaw=0.0)**

Команда движения к указанной точке с заданной ориентацией.

**Параметры:**

* x (float): X-координата цели
* y (float): Y-координата цели
* yaw (float, по умолчанию 0.0): Желаемая ориентация в радианах

**Возвращает:** None

**Пример использования:**

# Движение к точке (1.0, 2.0) без изменения ориентации  
bot.navigate(1.0, 2.0)  
  
# Движение к точке с поворотом на 90 градусов  
import math  
bot.navigate(1.0, 2.0, math.pi/2)

**navigate\_path(waypoints)**

Навигация по заданному маршруту из нескольких точек.

**Параметры:**

* waypoints (List[Tuple[float, float, float]]): Список точек маршрута (x, y, yaw)

**Возвращает:** None

**Пример использования:**

# Создание маршрута из трёх точек  
path = [  
 (1.0, 0.0, 0.0), # Первая точка  
 (1.0, 1.0, math.pi/2), # Вторая точка с поворотом  
 (0.0, 1.0, math.pi) # Третья точка с поворотом на 180°  
]  
  
bot.navigate\_path(path)

**6. Контроль выполнения движения**

**point\_reached()**

Проверяет статус достижения целевой точки.

**Возвращает:**

* int: Код статуса (обычно 0 - не достигнута, 1 - достигнута)

**Пример использования:**

bot.navigate(2.0, 3.0)  
  
while bot.point\_reached() == 0:  
 print("Движемся к цели...")  
 time.sleep(0.5)  
  
print("Цель достигнута!")

**path\_len()**

Возвращает длину текущего планируемого пути.

**Возвращает:**

* float: Длина пути в метрах

**Пример использования:**

bot.navigate(5.0, 5.0)  
path\_length = bot.path\_len()  
print(f"Планируемая длина пути: {path\_length:.2f} м")

**7. Работа с картами**

**save\_map(name)**

Сохраняет текущую карту под указанным именем.

**Параметры:**

* name (str): Имя карты для сохранения

**Возвращает:** None

**Пример использования:**

bot.save\_map("office\_map\_v1")  
print("Карта сохранена")

**load\_map(name, pose=None)**

Загружает сохранённую карту и опционально устанавливает позицию робота.

**Параметры:**

* name (str): Имя карты для загрузки
* pose (Optional[Tuple[float, float, float]]): Начальная позиция (x, y, yaw)

**Возвращает:** None

**Пример использования:**

# Загрузка карты без установки позиции  
bot.load\_map("office\_map\_v1")  
  
# Загрузка карты с установкой начальной позиции  
bot.load\_map("office\_map\_v1", pose=(0.0, 0.0, 0.0))

**8. Управление джойстиком**

**joy\_button(button\_name)**

Проверяет состояние кнопки джойстика.

**Параметры:**

* button\_name (str): Название кнопки

**Возвращает:**

* bool: True если кнопка нажата, False иначе

**Пример использования:**

if bot.joy\_button("A"):  
 print("Кнопка A нажата")  
  
if bot.joy\_button("start"):  
 print("Кнопка start нажата")

**joy\_axis(axis\_name)**

Получает значение оси джойстика.

**Параметры:**

* axis\_name (str): Название оси

**Возвращает:**

* float: Значение оси (обычно от -1.0 до 1.0)

**Пример использования:**

left\_stick\_x = bot.joy\_axis("left\_stick\_x")  
left\_stick\_y = bot.joy\_axis("left\_stick\_y")  
  
print(f"Левый стик: X={left\_stick\_x:.2f}, Y={left\_stick\_y:.2f}")

**9. Управление манипулятором**

**grip()**

Выполняет операцию захвата/зажима.

**Возвращает:** None

**Особенности:**

* Операция занимает около 8 секунд
* Метод блокирует выполнение на время операции

**Пример использования:**

print("Начинаем захват...")  
bot.grip() # Блокируется на 8 секунд  
print("Захват завершён")

**rize()**

Выполняет операцию отпускания груза.

**Возвращает:** None

**Особенности:**

* Операция занимает около 8 секунд
* Метод блокирует выполнение на время операции

**Пример использования:**

print("Начинаем подъём...")  
bot.rize() # Блокируется на 8 секунд  
print("Подъём завершён")

**10. Мониторинг и статистика**

**stats()**

Получает статистику работы клиента.

**Возвращает:**

* Dict[str, float]: Словарь со статистикой
  + sent: Количество отправленных сообщений
  + received: Количество полученных сообщений
  + cps: Команд в секунду (commands per second)

**Пример использования:**

statistics = bot.stats()  
print(f"Отправлено: {statistics['sent']}")  
print(f"Получено: {statistics['received']}")  
print(f"Скорость: {statistics['cps']:.2f} команд/сек")

**Обработка ошибок**

Клиент автоматически обрабатывает сетевые ошибки и переподключения:

* **Таймауты**: По умолчанию 3 секунды ожидания ответа
* **Повторные попытки**: До 5 попыток при потере соединения
* **Автоматическое переподключение**: При разрыве соединения

**Возможные исключения**

try:  
 bot.navigate(1.0, 1.0)  
except ConnectionError:  
 print("Сервер недоступен")  
except TimeoutError:  
 print("Превышено время ожидания ответа")

**Полный пример использования**

from nav\_api\_ws import NavBot  
import time  
import math  
  
# Подключение к роботу  
bot = NavBot(host="192.168.0.10")  
  
try:  
 # Ожидание готовности  
 print("Ожидание готовности робота...")  
 bot.wait\_until\_ready(30.0)  
   
 # Получение текущей позиции  
 x, y, yaw = bot.get\_pose()  
 print(f"Текущая позиция: ({x:.2f}, {y:.2f}, {yaw:.2f})")  
   
 # Проверка доступности цели  
 target\_x, target\_y = 2.0, 1.0  
 available, distance = bot.check\_target(target\_x, target\_y)  
   
 if available:  
 print(f"Цель доступна на расстоянии {distance:.2f}м")  
   
 # Движение к цели  
 bot.navigate(target\_x, target\_y, math.pi/4)  
   
 # Ожидание достижения цели  
 while bot.point\_reached() == 0:  
 print("Движемся к цели...")  
 time.sleep(1.0)  
   
 print("Цель достигнута!")  
   
 # Сохранение текущей карты  
 bot.save\_map("exploration\_result")  
   
 else:  
 print(f"Цель недоступна (расстояние: {distance:.2f}м)")  
   
 # Получение статистики  
 stats = bot.stats()  
 print(f"Статистика: {stats}")  
   
except Exception as e:  
 print(f"Ошибка: {e}")

**Тестирование и запуск API**

**1. Установка зависимостей**

Для работы с NavBot WebSocket API и тестовыми скриптами необходимо установить следующие пакеты Python:

# bash   
pip install websockets asyncio

**2. Структура папки robot\_api**

В папке robot\_api находятся следующие файлы:

* rf\_bot\_api.py - основной клиентский API
* test\_server.py - тестовый сервер для эмуляции робота
* demo\_navbot\_api.py - демонстрационный скрипт с примерами использования
* rf\_bot\_api\_docs.docx - данная документация

**3. Запуск тестовой среды**

Шаг 1: Запуск тестового сервера

# bash  
python test\_server.py

Сервер запустится на ws://localhost:8765 и будет выводить в консоль:

Сервер запущен: ws://0.0.0.0:8765

Ожидание подключений...

Шаг 2: Запуск демо-скрипта

# bash  
python demo\_navbot\_api.py

Скрипт последовательно протестирует все методы API и выведет результаты в консоль.

**4. Рекомендации по работе в VSCode**

1. Откройте папку проекта:
   * Запустите VSCode
   * Выберите "File" > "Open Folder"
   * Укажите путь к папке robot\_api
2. Настройка окружения:
   * Убедитесь, что в VSCode выбран правильный интерпретатор Python (Ctrl+Shift+P > "Python: Select Interpreter")
   * Рекомендуется создать виртуальное окружение:

# bash   
python -m venv .venv  
.\.venv\Scripts\activate # Windows  
source .venv/bin/activate # Linux/macOS  
pip install websockets

1. **Запуск и отладка:**
   * Для тестового сервера: откройте test\_server.py и нажмите F5
   * Для клиента: откройте demo\_navbot\_api.py и нажмите F5
   * Используйте точки останова для отладки
2. **Полезные расширения VSCode:**
   * Python (Microsoft)
   * Pylance
   * REST Client
   * Code Runner
   * Python Docstring Generator

**Заключение**

NavBot WebSocket Client API предоставляет полнофункциональный интерфейс для удалённого управления роботом. Все операции выполняются синхронно, что упрощает написание логики управления. Встроенные механизмы обработки ошибок и переподключения обеспечивают надёжность работы в сетевой среде.