# **API OCPP con ASYNC WEBSOCKET**

The Open Charge Point Protocol (OCPP) is a network protocol for communication between electric vehicle chargers and a central backoffice system.

## **Funcionality**

This library is the implementation of OCPP developed and used by NewMotion, one of Europe's largest Electric Vehicle Charge Point Operators.

This library only implements the network protocol. That is, it provides data types for the OCPP messages, remote procedure call using those request and response messages, and error reporting about those remote procedure calls. It does not provide any actual handling of the message contents.

The library is designed with versatility in mind. OCPP comes in 2 versions (1.6 and 2.0), two transport variants WebSocket/JSON, and two roles ("Charge Point" emulated and "Central System"). This library will help you with 1.6 over JSON.

## **How to use**

### **setup**

The library is divided into two separate modules so applications using it won't get too many dependencies dragged in. Those are:

* ocpp-j-api: high-level interface to OCPP-J connections
* ocpp-json: serialization of OCPP messages to/from JSON

### **Starting**

*These instructions will allow you to get a copy of the project running on your local machine for development and testing purposes.*

### **Instructions:**

1. you must to have python 3.7 running on virtual environment

* versions Debian, for example Ubuntu, use APT.

$ sudo apt-get install python3.7

* in Red Hat, fedora and Centos use yum.

$ sudo yum install python37

* in SUSE, use zypper.

$ sudo zypper install python3-3.7

for verify that python is running successfully, open the terminal or shell and execute the next command.

$ python3 --versionPython 3.7.3

See **Deployment** to know how to deploy the project.

### **Requirements**

#### **Python version**

*We recommend using the latest version of Python and django. this API supports Python 3.7, Django 3 and newers.*

#### **Dependencies**

aioredis==1.3.1

asgiref==3.3.1

astroid==2.4.2

async-timeout==3.0.1

attrs==20.3.0

autobahn==20.12.3

Automat==20.2.0

certifi==2020.12.5

cffi==1.14.4

channels==3.0.3

channels-redis==3.2.0

chardet==4.0.0

colorama==0.4.4

constantly==15.1.0

cryptography==3.3.1

daphne==3.0.1

defusedxml==0.6.0

Django==3.1.5

django-allauth==0.44.0

django-filter==2.4.0

django-rest-auth==0.9.5

django-restframework==0.0.1

djangorestframework==3.12.2

h2==3.2.0

hiredis==1.1.0

hpack==3.0.0

hyperframe==5.2.0

hyperlink==21.0.0

idna==2.10

importlib-metadata==3.4.0

incremental==17.5.0

isort==5.7.0

jsonschema==3.2.0

lazy-object-proxy==1.4.3

Markdown==3.3.3

mccabe==0.6.1

msgpack==1.0.2

oauthlib==3.1.0

ocpp==0.8.1

priority==1.3.0

psycopg2==2.8.6

pyasn1==0.4.8

pyasn1-modules==0.2.8

pycparser==2.20

PyHamcrest==2.0.2

PyJWT==2.0.1

pylint==2.6.0

pyOpenSSL==20.0.1

pyrsistent==0.17.3

python3-openid==3.2.0

pytz==2020.5

requests==2.25.1

requests-oauthlib==1.3.0

service-identity==18.1.0

six==1.15.0

sqlparse==0.4.1

toml==0.10.2

Twisted==21.2.0

txaio==20.12.1

typed-ast==1.4.2

typing-extensions==3.7.4.3

urllib3==1.26.3

wrapt==1.12.1

zipp==3.4.0

zope.interface==5.2.0

*You can save all these dependencies in a file requirements.txt in the folder's project*

## **Installation** 🔧

#### **Virtual environments**

Use a virtual environment to manage the dependencies for your project, both in development and in production.

What problem does a virtual environment solve? The more Python projects you have, the more likely it is that you need to work with different versions of Python libraries, or even Python itself. Newer versions of libraries for one project can break compatibility in another project.

Virtual environments are independent groups of Python libraries, one for each project. Packages installed for one project will not affect other projects or the operating system’s packages.

Python comes bundled with the venv module to create virtual environments.

$ mkdir API\_OCPP\_PORT$ cd API\_OCPP\_PORT

$ python3 -m venv machine\_ws

##### **Activate the environment**

activate your enviroment

$ source machine\_ws/bin/activate

Your shell prompt will change to show the name of the activated environment.

#### **Installing dependencies**

If you created the file *requirements.txt* with the **dependencies**, in your shell write:

(machine\_ws)$ pip install -r requirements.txt

## Running the tests ⚙️

In a browser Chromium install a client webSocket extension, it's suggered *Simple WebSocket Client*. you can download [here](https://chrome.google.com/webstore/detail/simple-websocket-client/pfdhoblngboilpfeibdedpjgfnlcodoo).

after installing it:

1. Enter the URL for your Web Socket server.
2. Click Open.
3. Input request text, then click Send.
4. The extension show response messages.

*Note: OCPP uses a camelCase naming scheme for the keys in the payload. Python, on the other hand, uses snake\_case. Therefore this ocpp package converts all keys in messages from camelCase to snake\_case and vice versa to make sure you can write Pythonic code. Now start the websocket server again and connect a client to it as you did before. If the client is connected send this BootNotification to the central system:*

*Note: The charge point's connection URL contains the charge point identity so that the Central System knows which charge point a Websocket connection belongs to.*

##### Example the url:

ws://localhost:8000/ws/charger/chager\_name/

**Insert in the text box:**

[2, "12345", "BootNotification", {"chargePointVendor": "TecnoBot Developers", "chargePointModel": "myCharger"}]

The server should respond and the you should see something like this: **Response:**

[3, "12345", {"currentTime": "2021-05-16T11:09:01.354678", "interval": 10, "status": "Accepted"}]`

you must to validate with a charger enabled for this in the database of chargers, please review your credentials for have access at the proof points

## Deploying 📦

Start the development server:

if you is executing de application for first time insert the next command

$ python manage.py migrateOperations to perform: Apply all migrations: admin, auth, contenttypes, sessionsRunning migrations: Applying contenttypes.0001\_initial... OK Applying auth.0001\_initial... OK Applying admin.0001\_initial... OK Applying admin.0002\_logentry\_remove\_auto\_add... OK Applying admin.0003\_logentry\_add\_action\_flag\_choices... OK Applying contenttypes.0002\_remove\_content\_type\_name... OK Applying auth.0002\_alter\_permission\_name\_max\_length... OK Applying auth.0003\_alter\_user\_email\_max\_length... OK Applying auth.0004\_alter\_user\_username\_opts... OK Applying auth.0005\_alter\_user\_last\_login\_null... OK Applying auth.0006\_require\_contenttypes\_0002... OK Applying auth.0007\_alter\_validators\_add\_error\_messages... OK Applying auth.0008\_alter\_user\_username\_max\_length... OK Applying auth.0009\_alter\_user\_last\_name\_max\_length... OK Applying auth.0010\_alter\_group\_name\_max\_length... OK Applying auth.0011\_update\_proxy\_permissions... OK Applying auth.0012\_alter\_user\_first\_name\_max\_length... OK Applying sessions.0001\_initial... OK

$ python3 manage.py runserver 0.0.0.0:8000

Every consumer instance has an automatically generated unique channel name, and so can be communicated with via a channel layer.

In our chat application we want to have multiple instances of Consumer(chargers) in the same room communicate with each other. To do that we will have each Consumer add its channel to a group whose name is based on the room name. That will allow Consumers (chargers) to transmit messages to all other Consumer(controlCharger) in the same room.

In this app is use a channel layer that uses Redis as its backing store. To start a Redis server on port 6379, run the following command:

$ docker run -p 6379:6379 -d redis:5

We need to install channels\_redis so that Channels knows how to interface with Redis. Run the following command:

$ python3 -m pip install channels\_redis

Before we can use a channel layer, we must configure it. Edit the port\_ocpp/settings.py file and add a CHANNEL\_LAYERS setting to the bottom. It should look like:

# port\_ocpp/settings.py

# ChannelsASGI\_APPLICATION = 'port\_ocpp.asgi.application'CHANNEL\_LAYERS = { 'default':

{

'BACKEND': 'channels\_redis.core.RedisChannelLayer',

'CONFIG': {

"hosts": [('127.0.0.1', 6379)], }, },

}

Let’s make sure that the channel layer can communicate with Redis. Open a Django shell and run the following commands:

$ python3 manage.py shell

>>> import channels.layers

>>> channel\_layer = channels.layers.get\_channel\_layer()

>>> from asgiref.sync

import async\_to\_sync

>>> async\_to\_sync(channel\_layer.send)('test\_channel', {'type': 'hello'})

>>>async\_to\_sync(channel\_layer.receive)('test\_channel'){'type': 'hello'}

## Built with 🛠️

* [django](https://www.djangoproject.com/) - The framework web used
* [ocpp](https://www.openchargealliance.org/) - Driver, schematics and docs

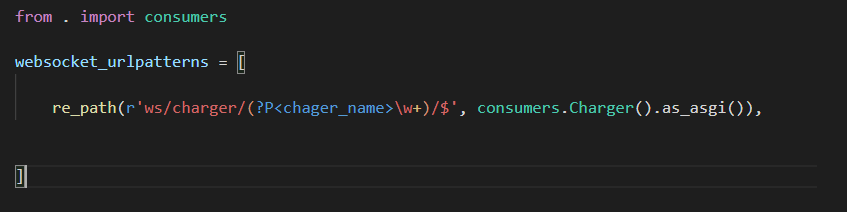
## Information of class:

Here you will find the endpoints more important of the application

### **Api´s connection end-point controller**Urls.py

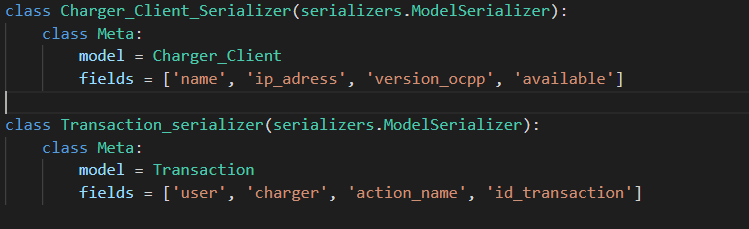
In this model you can access from your client Postman, Insomnia or favorite browser. You Remember access to system first logging as user authorized and so follow executing all functions of the API

### **Charger´s end-point connection**



In this connection´s point, the name charger is finded in the end of url, you must to sure than the name charge end in “chargerName/”

### **Serializers of control on server**



**Model charger controller in database**

## **Proof application.**

### ***Create users***

#### Link endpoint

<http://51.81.86.133:8000/api/v1/auth/registration/>

#### Body json

{

"username": "myUserNew",

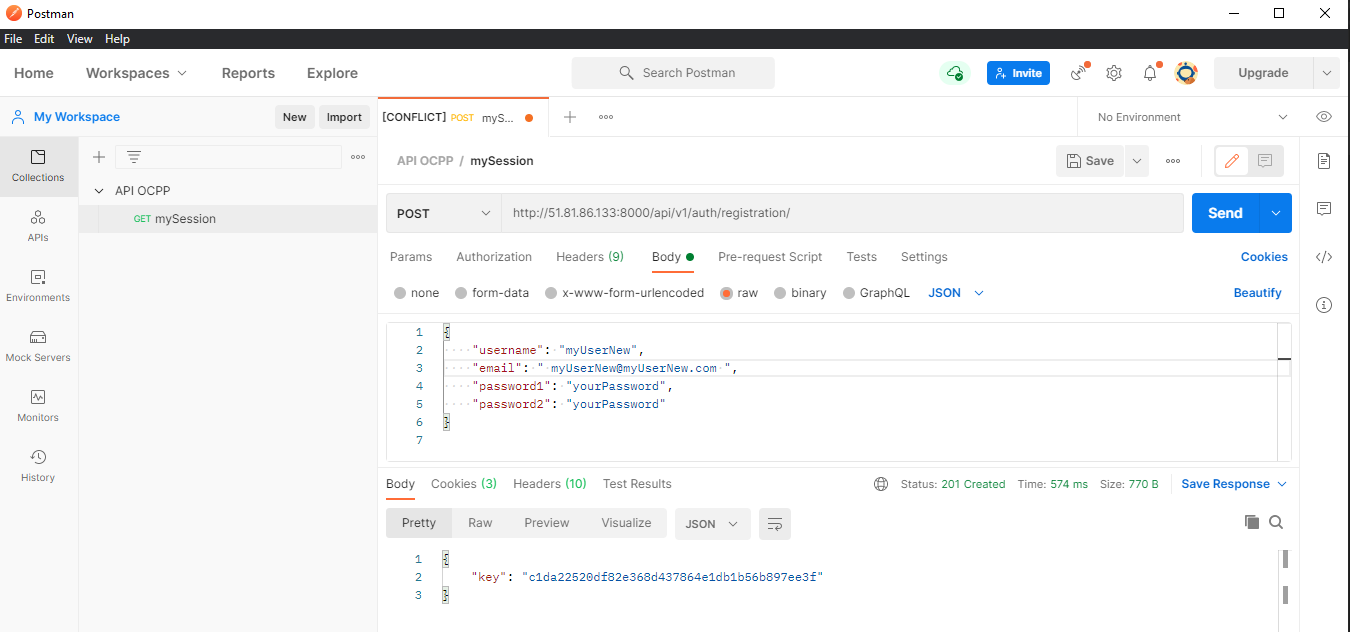
"email": " myUserNew ",

"password1": "yourPassword",

"password2": "yourPassword"

}

#### Method POST



### **Logging users**

#### Link endpoint

<http://51.81.86.133:8000/api/v1/auth/login/>

#### Body json

{

"username": "",

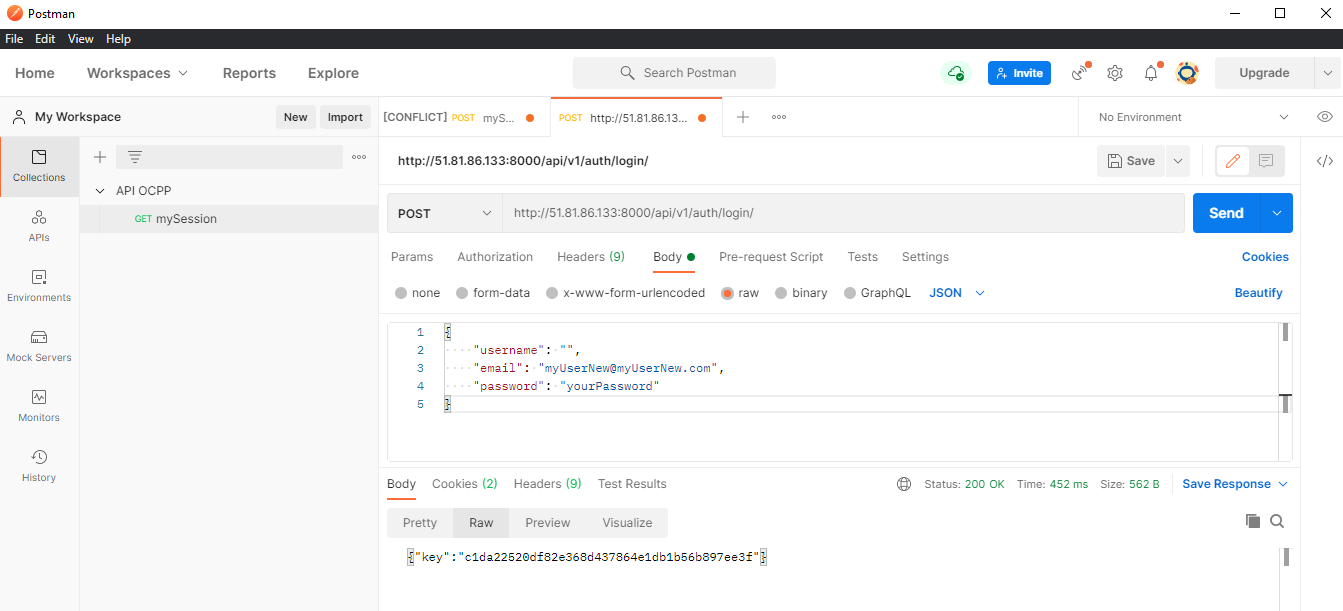
"email": "myUserNew@myUserNew.com",

"password": "yourPassword"

}

Check the credentials and return the REST Token, if the credentials are valid and authenticated. Calls Auth login method to register User ID in the login session framework Accept the following POST parameters: username, password, Return the Token Object's key.

#### Method POST



### **Add/get Chargers**

#### Link endpoint

<http://51.81.86.133:8000/api/v1/charger/>

#### Body json

    {

        "name": "PC\_remote",

        "ip\_adress": "0.0.0.0",

        "version\_ocpp": "16",

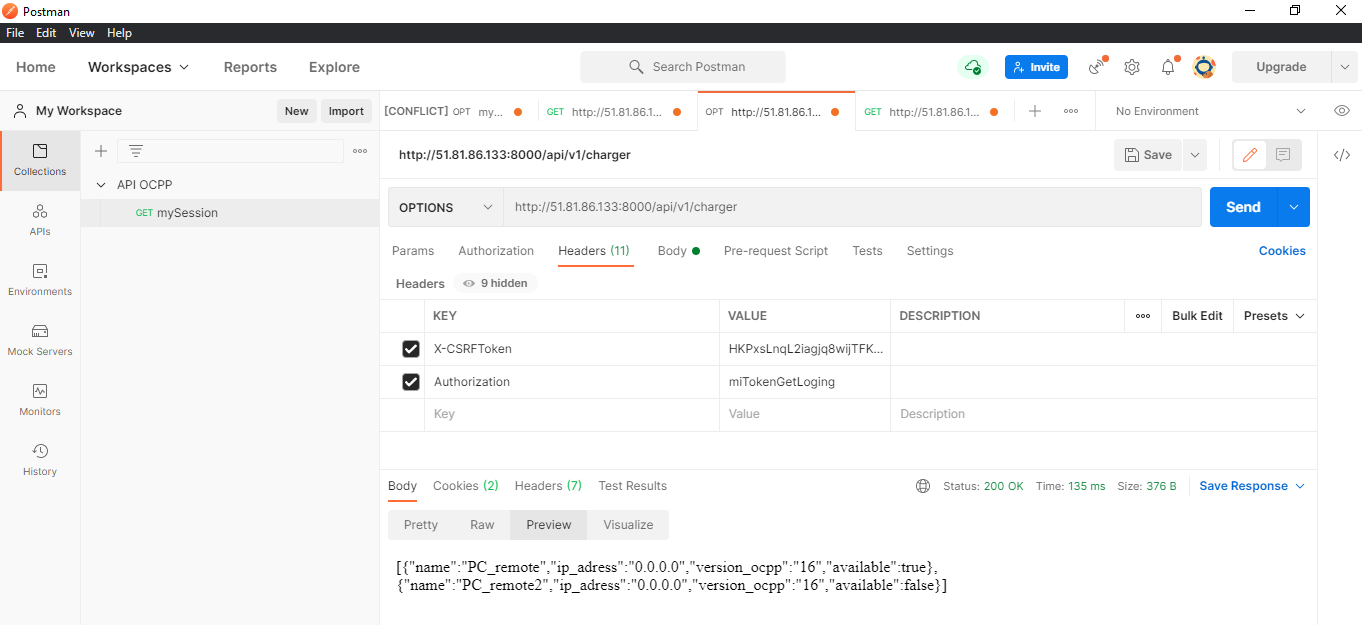
        "available": true

    }

You should know that to add or modify any charger the user must have an administrator role and use the POST method.

To enter remember that you must insert the CSRF token and the identification session in the header of your RestApi client (postman, insomnia)

#### Method POST /GET



### ***Modify Charger***

#### Link endpoint

<http://51.81.86.133:8000/api/v1/charger/2/>

#### Body json

    {

        "name": "PC\_remote",

        "ip\_adress": "8.8.8.8",

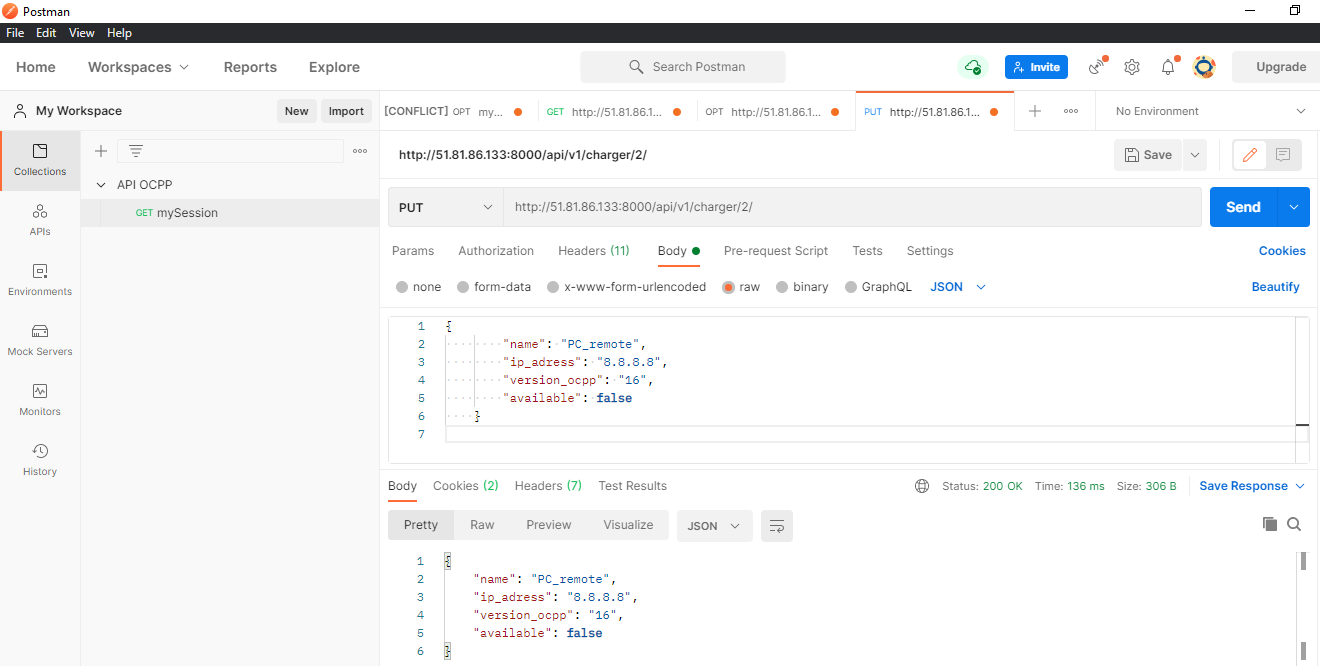
        "version\_ocpp": "16",

        "available": false

    }

To enter remember that you must insert the CSRF token and the identification session in the header of your RestApi client (postman, insomnia)

#### Method DEL /PATCH/PUT



### ***Start and Stop a transaction***

#### Link endpoint

http://51.81.86.133:8000/api/v1/charger/2/

#### Body json

{

"user": 5,

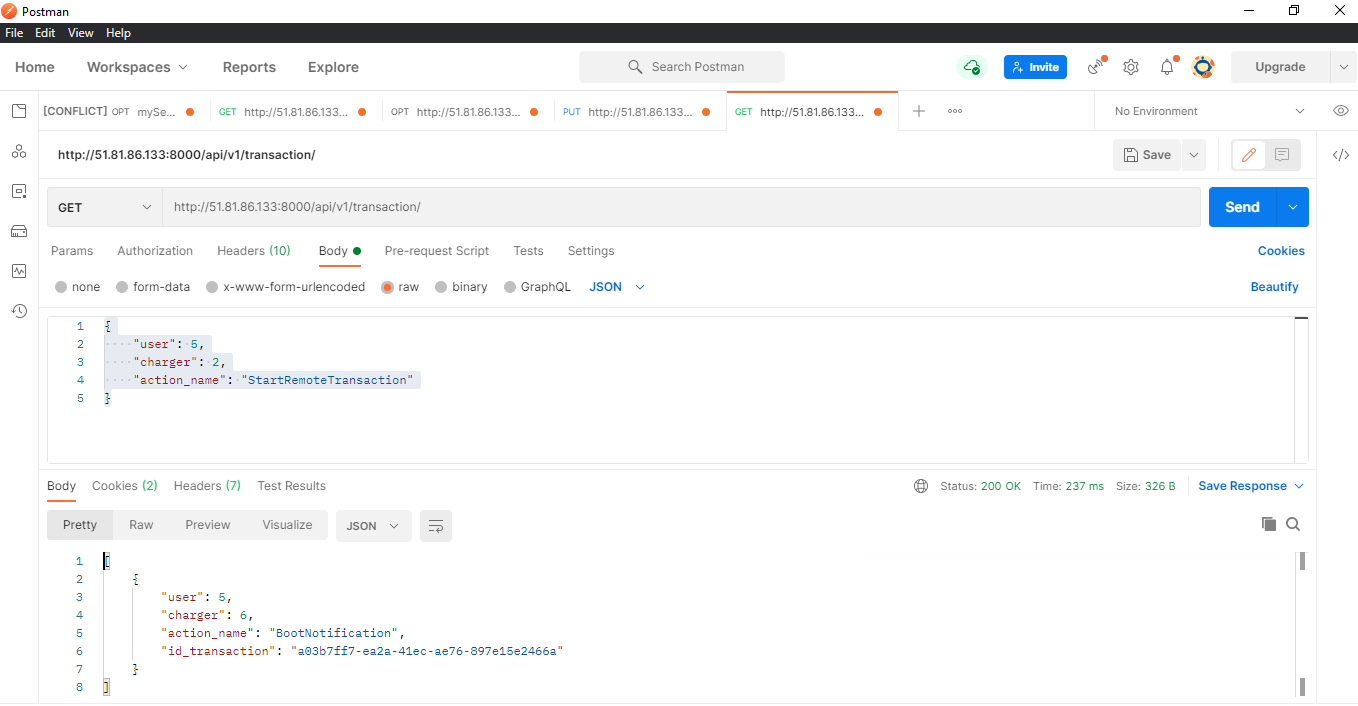
   "charger": 2,

"action\_name": "StartRemoteTransaction"

}

The different action name are <<StartRemoteTransaction>> and <<StopRemoteTransaction>>, the petition is stored in the database when it’s received a response from the charger

* 1. *Method POST*



### ***Start in db***

#### Link endpoint

http://51.81.86.133:8000/admin/

#### Body json

{

"user": admin,

   "pass": “YourPass”,

}

