
Innopolis University

Universitetskaya St, 1, Innopolis,
Respublika Tatarstan,
420500

Electric Autonomous Vehicles

July-August 2019



Table of Contents

Table of Contents	2
GLOSSARY	4
Chapter 1. Project Overview	5
Introduction	5
GOALS	6
SIMULATIONS	6
SUMO	6
How to install	6
Running SUMO on AWS	8
Pre-requisites	10
AWS instance creation	10
Virtual machine creation using VMware	15
Connecting to a virtual machine via SSH	16
GUI applications remote access	16
Sharing AWS instance with another AWS account	19
Network communication	21
Charging Experiment	22
Proof of concept	22
Architecture	22
Results	23
Chapter 2. Implementing the decision logic	24
Summary	24
Scenario steps of the charging process	24
System Database	27
Entity-Relationship Diagram	27
Web application	27
Car owner registration	27
Charging station selection	27
Project repository	28
Link of the project repository	28
Important remarks to consider	29

GLOSSARY

CS refers to the electrical charging stations that charges cars

EAV Electrical Autonomous Vehicles, the main subject of the project

IOTA Internet over the air

Instance By instances we mean cars and vehicles which will communicate with one another

M2M Machine to machine communication (Without interference of humans)

SUMO Simulation of Urban mobility software application, that tracks traffic and manages vehicle communication

EVSE The electric vehicles smart tool which will measure the amount of electricity each car wants.

Chapter 1. Project Overview

Introduction

The proliferation of electric vehicles (EVs) has spurred the research interest in technologies associated with it, for instance batteries, and charging mechanisms. Moreover, the recent advancements in autonomous cars also encourage the enabling technologies to integrate and provide holistic applications.

To this end, one key requirement for EVs is to have an efficient, secure, and scalable infrastructure and framework for charging, billing, and auditing. However, the current manual charging systems for EVs may not be applicable to the autonomous cars that demand new, automatic, secure, efficient, and scalable billing and auditing mechanism

Our vision demands investment on the creation of Electric Roads (ER), meaning roads with electric strips that would allow charging on-the-go. In such a scenario, EAV would be able to charge without stopping at the station and without the need of any human interaction. While the wireless method provides numerous benefits, and we consider it to be a more suitable long-term solution for EAV charging, its mass-scale deployment is not yet feasible at the moment. Alternatively, regular Charging Stations (CSs) should be considered.

When a vehicle needs to be charged, the application embedded inside the electric vehicle starts searching for the nearest charging station. Afterwards, a secure payment channel is established and both parties (The Car and the Station) transfer equal amount of IOTA into a multi signature address controlled by both parties, So the car gets the electricity it needs and the Charging station get the required amount of money.

GOALS

The project aims at developing a large-scale framework and attract the attention of funding agencies, the scientific community and industrial world. In particular we aim at:

1. Computer graphical simulation for the M2M charging system
2. 3D physical model for the system
3. Full scale in-vitro development
4. Propose the project to the authorities

SIMULATIONS

In order to make experimental investigation in the lab, we have used SUMO (Simulation of Urban mobility) to aid in vehicular communication and to manage traffic. In other words, Each vehicle can communicate with a number of other vehicles and CSs located in the same area limited by WIFI connectivity range.

SUMO

How to install

First we need to install SUMO package in our Ubuntu instance, so we will follow the following steps:

1. Install all of the required tools and libraries (it is recommended to use cmake)
2. Get the source code and defining the *SUMO_HOME* environment variable
3. Build the SUMO binaries

Each of these steps is described in more detail below.

1. Installing all the required tools and libraries

To be able to run the last version of SUMO in your device/instance all the following libraries are prerequisites:

cmake, python, g++ ,libxerces-c-dev ,libfox-1.6-dev ,libgdal-dev, libproj-dev, libgl2ps-dev, swig

Simply enter the following command to your terminal:

```
$] sudo apt-get install cmake python g++ libxerces-c-dev libfox-1.6-dev libgdal-dev  
libproj-dev libgl2ps-dev swig
```

2. Getting the source code

- We need to download the latest version of SUMO, which is recommended to be downloaded from the official Github repository. The following commands should be issued:

```
$] git clone --recursive https://github.com/eclipse/sumo
```

```
$] cd sumo
```

```
$] git fetch origin refs/replace/*:refs/replace/*
```

```
$] pwd
```

- Now what we need to do is to define SUMO_HOME environment variable as follows:

To permanently add the SUMO_HOME environment variable so we don't need to add it every time we reboot, simply write the following commands:

```
$] gedit ~/.profile
```

Now add the following line as the last line of the document

```
export SUMO_HOME="/home/<user>/sumo-<version>"
```

3. Building the SUMO binaries with cmake

To build with cmake version 3 or higher is required. Create a build folder for cmake (in the sumo root folder)

```
$] cd $PATH_TO_SUMO_ROOT_FOLDER
```

```
$] mkdir build/cmake-build
```

```
$] cd build/cmake-build
```

To build sumo with the full set of available options just like GDAL and OpenSceneGraph support (if the libraries are installed) just run: **[Recommended in our project]**

```
$] cmake ../..
```

Then we have to install the SUMO binaries as follows:

```
$] sudo make install
```

And finally remodify the SUMO_HOME directory usually (/usr/local/share/sumo):

```
$] gedit ~/.profile
```

Add the following line as the last line of the document

```
export SUMO_HOME="/usr/local/share/sumo"
```

Then Write the command:

```
$] sudo apt-get install sumo
```

Running SUMO on AWS

This is the second step in simulation after installing SUMO successfully, to run SUMO in our AWS instance the following steps should be followed:

1. Open your browser and open the link <https://www.openstreetmap.org>
2. From the left side banner choose “*Manually select a different area*”
3. Select a small area of the map and choose export, save the file as “Map.osm” in a new folder named Map and know its location for next steps

Now the next step will be done if you need to include (Shapes: Mountains, Green areas, etc..) to your model [**Recommended in our project**]

4. Download the following text file to your Map folder:

<https://drive.google.com/open?id=1S- LiSje3mtFu08hyE2erkeYrQCqQgl>

5. Create a new file and name it my_config_file.sumocfg, the configuration file will be used to simulate the cars and trips and cars, add the following text to it:

```
<configuration>
<input>
<net-file value="test.net.xml"/>
<route-files value="map.rou.xml"/>
<additional-files value="map.poly.xml"/>
</input>
```

```
<time>
<begin value="0"/>
<end value="2000"/>
</time>
</configuration>
```

6. Convert the map into a SUMO network using the command:

```
$] netconvert --osm-files map.osm -o test.net.xml
```

7. Then convert this SUMO network to include polygons (Shapes and buildings) as follows:

```
$] polyconvert --xml-validation never --net-file test.net.xml --osm-files map.osm  
--type-file typemap.xml -o map.poly.xml
```

8. Now run the python file stored in \$SUMO_ROOT/tools/randomTrips.py as follows by copying and pasting it to your Map folder and then type the following command

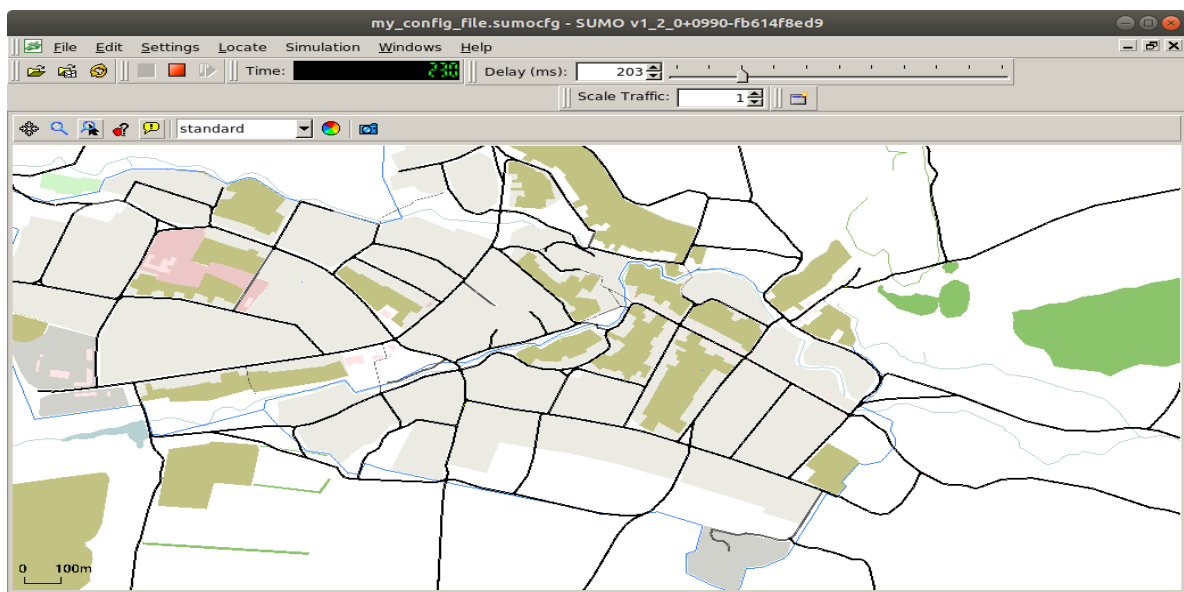
```
$] python randomTrips.py -n test.net.xml -e 100 -l
```

9. Now run the python file again using this code

```
$] python randomTrips.py -n test.net.xml -r map.rou.xml -e 100 -l
```

10. Now we are ready to use the .sumocfg file that we created at step 5

```
$] sumo-gui my_config_file.sumocfg
```



Finally, after successful Sumo setup a screen similar to this one should appear.

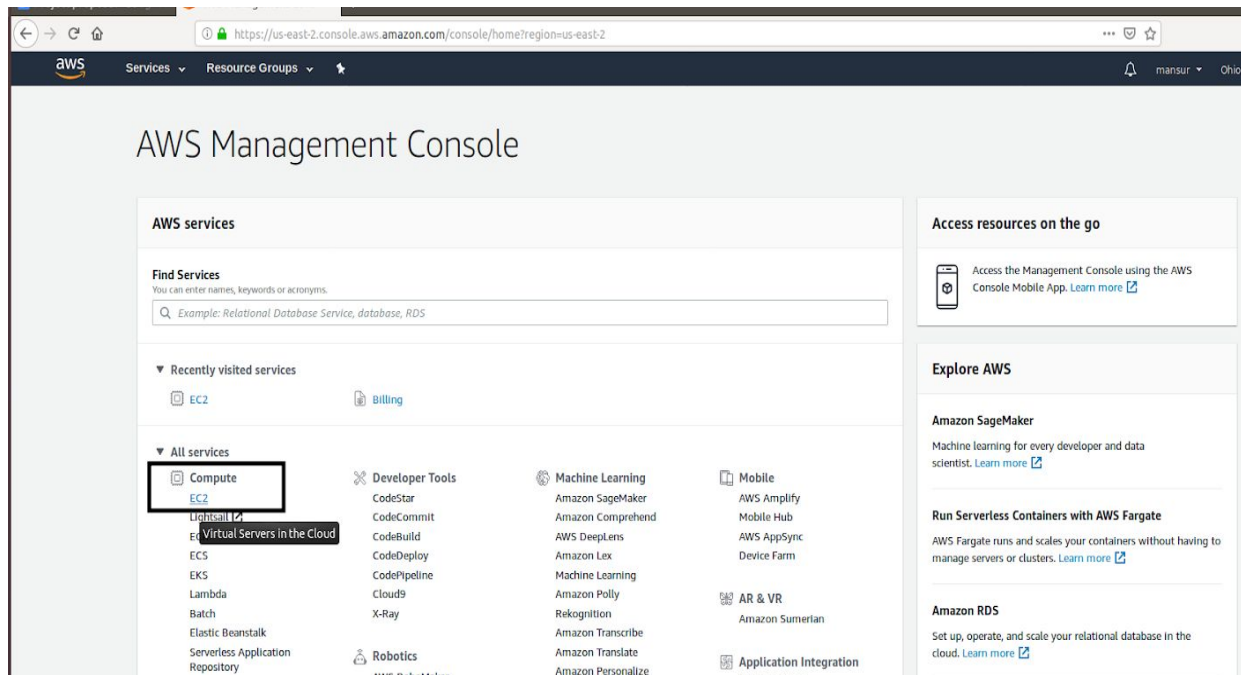
Pre-requisites

AWS instance creation

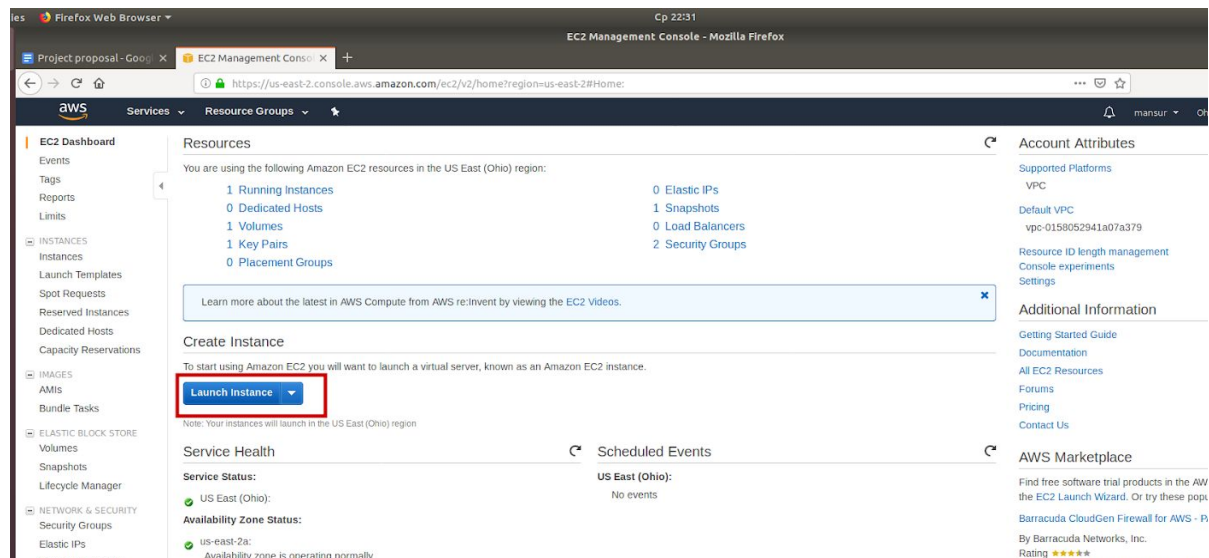
In order to investigate our research topic, instances (vehicles and charging stations) must be connected to each other so their local ip must be shared between them.

First we will discuss how to create an ec2 instance on AWS: *[You need to have an AWS account]*

1. From *Compute* choose *EC2*



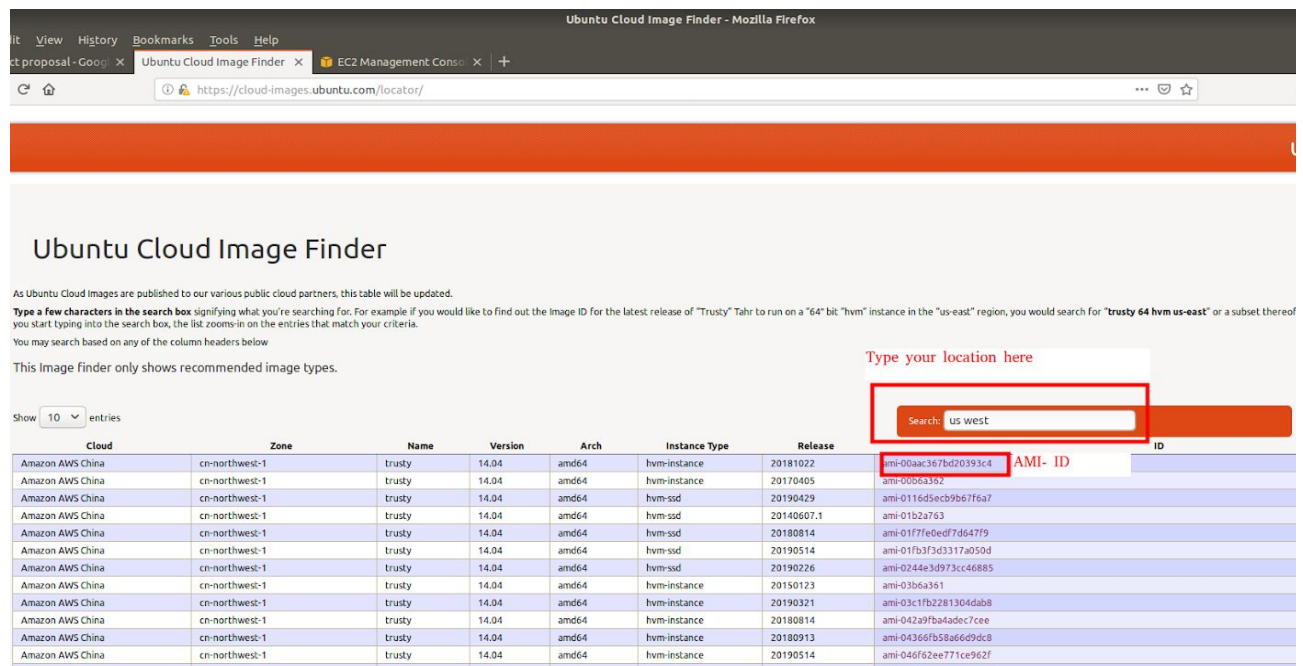
2. Click on *Launch instance*



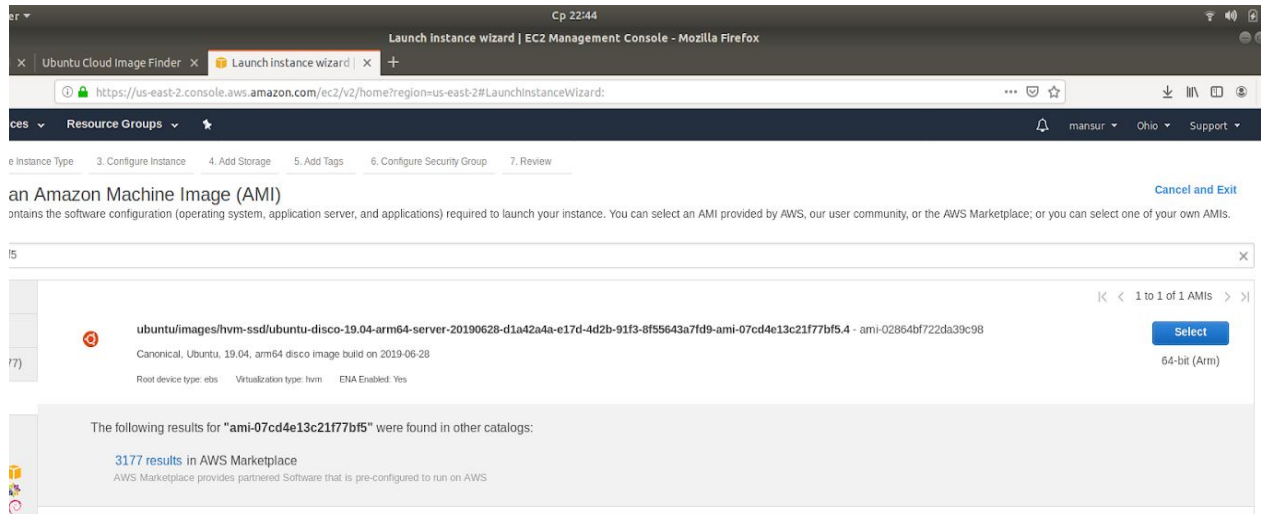
3. To get the official and most updated version of ubuntu (image) you want go to

<https://cloud-images.ubuntu.com/locator/>

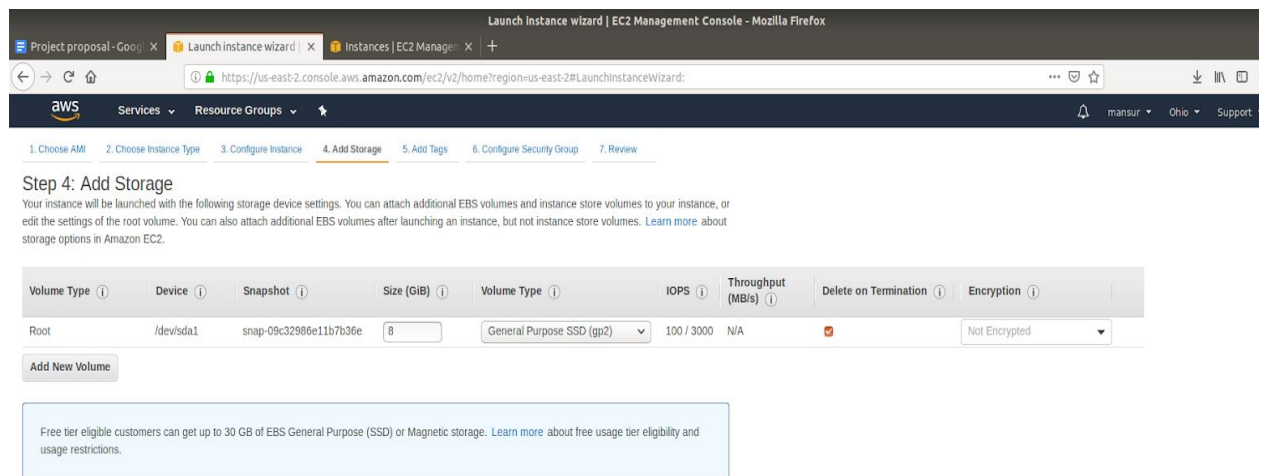
4. In the search box type “us-east” or “us-west” depending on your location, you will find your location in the top right corner beside your name, choose the image you want and copy the AMI - ID (The last column)



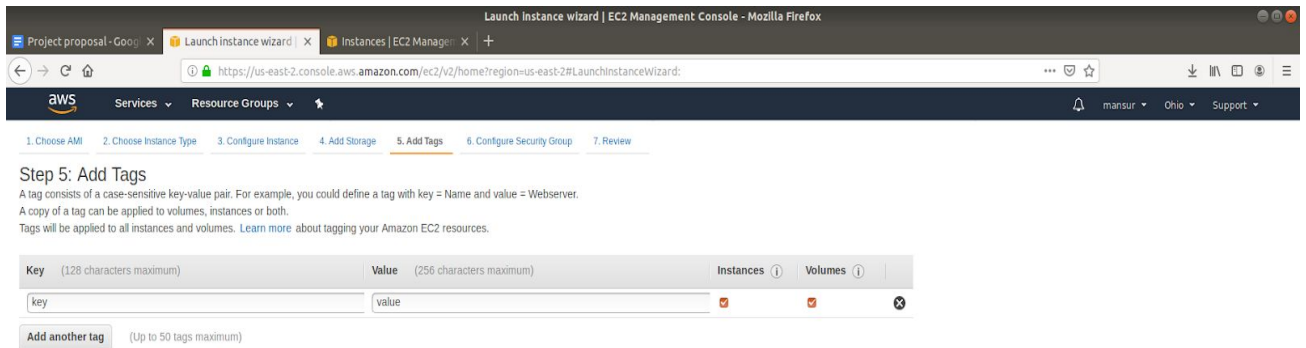
- Paste AMI-ID onto the search box of the AWS console after choosing *Community AMIs*, then Select the image that appeared to you



- Leave the selection as it is and click on *Configure instance details*, Then also here we will work all as default so click on *Add storage*, Afterwards Click *Add tags*



7. Add a value and a key for this tag and click on *Configure security group*



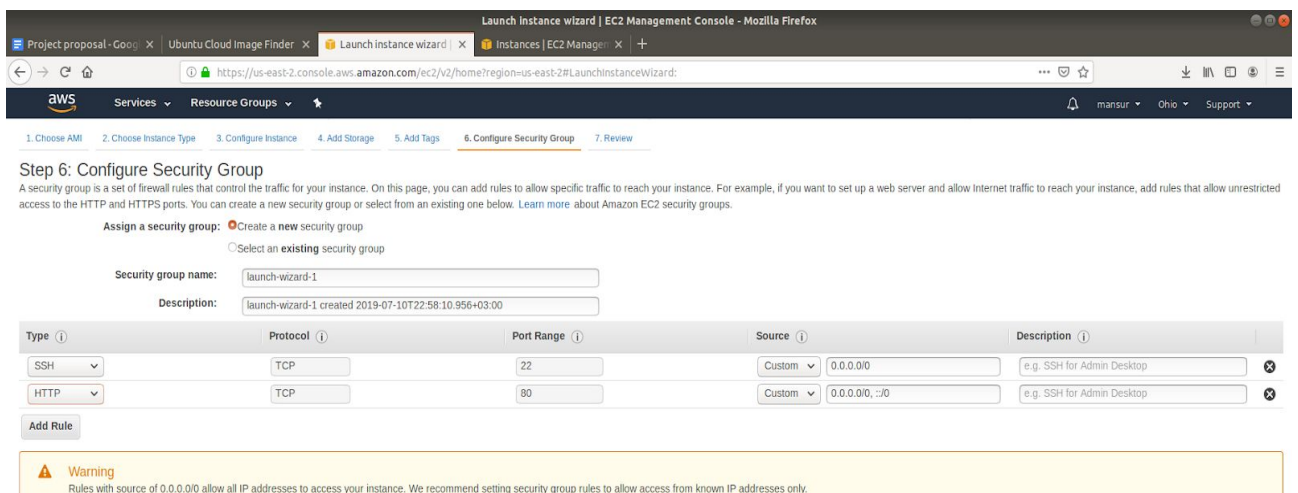
Step 5: Add Tags

A tag consists of a case-sensitive key-value pair. For example, you could define a tag with key = Name and value = Webserver. A copy of a tag can be applied to volumes, instances or both. Tags will be applied to all instances and volumes. [Learn more](#) about tagging your Amazon EC2 resources.

Key (128 characters maximum)	Value (256 characters maximum)	Instances	Volumes
key	value	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

[Add another tag](#) (Up to 50 tags maximum)

8. Write a name for the security group, and a description, then Click on *Add rule* and choose *HTTP*



Step 6: Configure Security Group

A security group is a set of firewall rules that control the traffic for your instance. On this page, you can add rules to allow specific traffic to reach your instance. For example, if you want to set up a web server and allow Internet traffic to reach your instance, add rules that allow unrestricted access to the HTTP and HTTPS ports. You can create a new security group or select from an existing one below. [Learn more](#) about Amazon EC2 security groups.

Assign a security group: ☒ Create a new security group
☐ Select an existing security group

Security group name:

Description:

Type	Protocol	Port Range	Source	Description
SSH	TCP	22	Custom 0.0.0.0/0	e.g. SSH for Admin Desktop
HTTP	TCP	80	Custom 0.0.0.0/0, :::0	e.g. HTTP for Admin Desktop

[Add Rule](#)

Warning
Rules with source of 0.0.0.0/0 allow all IP addresses to access your instance. We recommend setting security group rules to allow access from known IP addresses only.

9. Then click on *Review and launch* then *Launch*

10. From the list choose *create a key pair*, Download the Key pair file (.pem) to a known folder. Now the instance is created, so refresh the web page and in instances you will find it

11. Copy the public ip of the instance to use it to SSH (Connect) to the new instance

The screenshot shows the AWS Management Console for the 'Instances | EC2 Management Console - Mozilla Firefox' browser window. The URL is <https://us-east-2.console.aws.amazon.com/ec2/v2/home?region=us-east-2#instances:sort=instancetype>. The console displays a table of EC2 instances. The instance 'Dev-Server' is highlighted, and its IPv4 Public IP, 18.188.232.13, is highlighted with a red box and labeled 'Public IP'.

Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS (IPv4)	IPv4 Public IP	IPv6 IPs	Key Name	Monitoring
Dev-Server	i-0dec51c73f313fc98	t2.micro	us-east-2a	running	2/2 checks ...	None	ec2-18-188-232-13.us-e...	18.188.232.13	-	Innopolls-EAV	disab

Instance: i-0dec51c73f313fc98 (Dev-Server) Public DNS: ec2-18-188-232-13.us-east-2.compute.amazonaws.com

Virtual machine creation using VMware

1. Install VMware from the official website www.vmware.com
2. Create a virtual machine using VMware
 - a. From *Player*-> *File* choose *New virtual machine*
 - b. Choose installer disc image file (iso) and browse to your image

In case you must write username and password

Enter your machine's full name = 'Innopolis-EAV', user = 'innopolis' and password = 'innopolis' **[don't change *username* and *password*]**

- c. Choose the place you want your virtual machine to be stored to
- d. Click on split virtual machines into separate files
- e. Click on finish

Connecting to a virtual machine via SSH

You can securely connect to the instance you created *[You must have the keypair and the ip address]*

1. Move the key pair that you downloaded to your .ssh directory as follows, suppose your key pair file is named “keypair.pem” in “Downloads” folder:

```
mv Downloads/keypair.pem ~/.ssh/
```

2. Check the access permissions of the key pair:

```
ls -alh .ssh
```

3. Let the access permissions be rw for you and for the group by issuing the following command:

```
chmod 644 .ssh/keypair.pem
```

4. Now connect to the ip address of the instance using the following command:

```
ssh ubuntu@xx.xx.xxx.xxx -i .ssh/keypair.pem,
```

Where xx.xx.xxx.xxx is the ip address of your instance

GUI applications remote access

Now we suppose that we are connecting to another instance that we have its keypair file and its public IPv4 address, we will use X-Forwarding to remotely access GUI applications on the created AWS instance.

Remotely access GUI programs on an EC2 instance: **[*LINUX* method]**

1. First of all, SSH into the instance as explained above, then Install xauth program if you don't have it on the image machine

```
sudo apt-get install xauth
```

2. Now we must edit the client SSH by issuing the command

```
sudo nano /etc/ssh/ssh_config
```

-
3. Under host, get rid of the comment signs “#” before FORWARDX11 & FORWARDX11TRUSTED. And if it says no (to the right of it) replace it with yes
 4. Click CTRL+O (To write it out) Then Click CTRL+X to exit
 5. Now we must edit also the daemon so enter

```
sudo nano /etc/ssh/sshd_config
```

6. Scroll to the bottom till you find X11forwarding and X11Display, delete “#”
7. Click CTRL+O (To write it out) then Click CTRL+X to exit
8. Now we will connect to the instance by - X for forwarding

```
ssh -X ubuntu@xx.xx.xxx.xxx -i .ssh/keypair.pem
```

9. Now to install all the prerequisite libraries to access GUI write the command

```
Sudo apt-get -f install
```

10. Install the program you want to use and access it by typing its package name in the terminal or by using (service xxx start), where xxx is your application

Remotely access GUI programs on an EC2 instance: [**WINDOWS method**]

Prerequisites :

- Putty application.exe
- Puttygen.exe
- Key-pair file of the instance (.pem)
- Xming
- Xmingfonts

1. First download the latest version of Putty on your windows machine
 - a. Go to www.putty.org
 - b. Download the Putty.exe application so you don't have to run it

-
- c. Also download Puttygen.exe tool which we will use to convert the *.pem* key file
 2. Connect to your machine image using Putty, after converting the *.pem* file
 - a. Open puttygen.exe
 - b. Click *load* to load the current pem key file
 - c. Click on save private key
 - d. Now open putty.exe to connect to your machine image
 - i. Write down the public IPV address of your machine
 - ii. From the left choose Data and Write the username of your instance
 - iii. Then from the left again expand *SSH-> Auth* and import the pem file that you converted
 - iv. Under SSH choose *X11* and enable X11 forwarding
 - v. From the left choose session, name your session and save it so you can log to it after and click open
 - e. Now after saving the session connect to it
 3. Inside the terminal write the following command to edit the client server

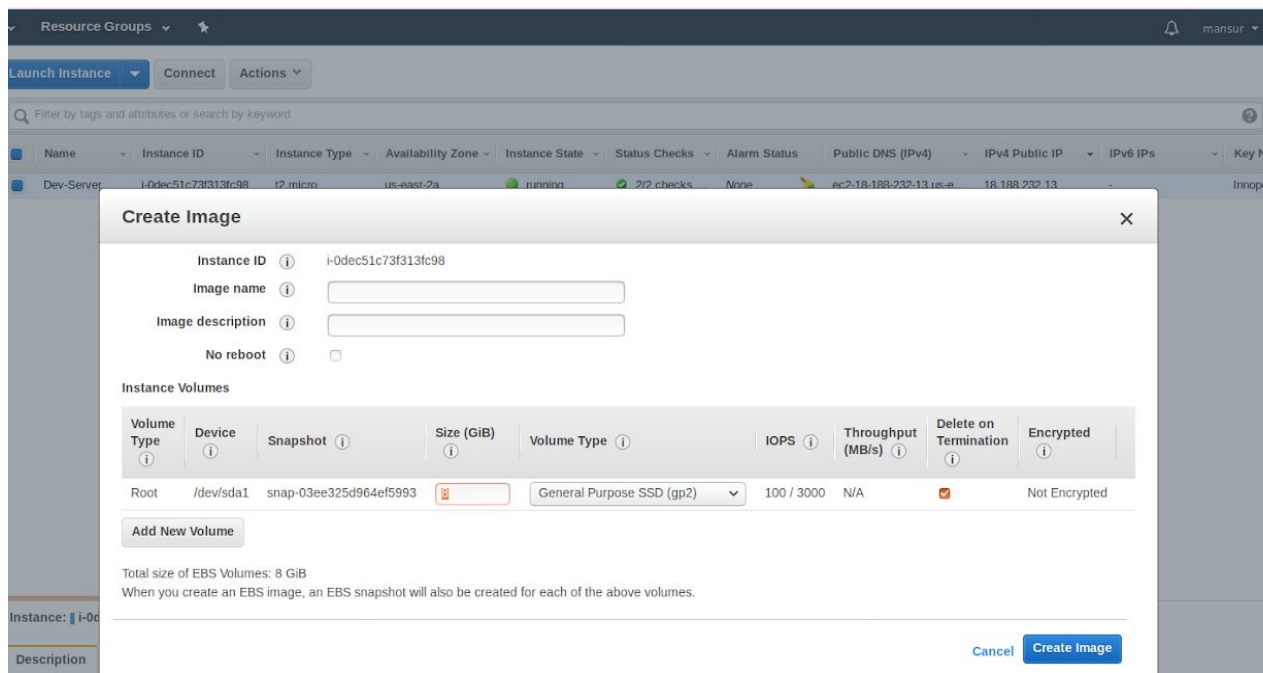
```
sudo nano /etc/ssh/ssh_config
```

- a. Under host edit the line ForwardX11 and write 'yes' to its right instead of no
4. Download Xming X server and Xming-fonts from sourceforge, Run them as administrator
5. Type the application you want to open in terminal followed by & to open as a separate process

Sharing AWS instance with another AWS account

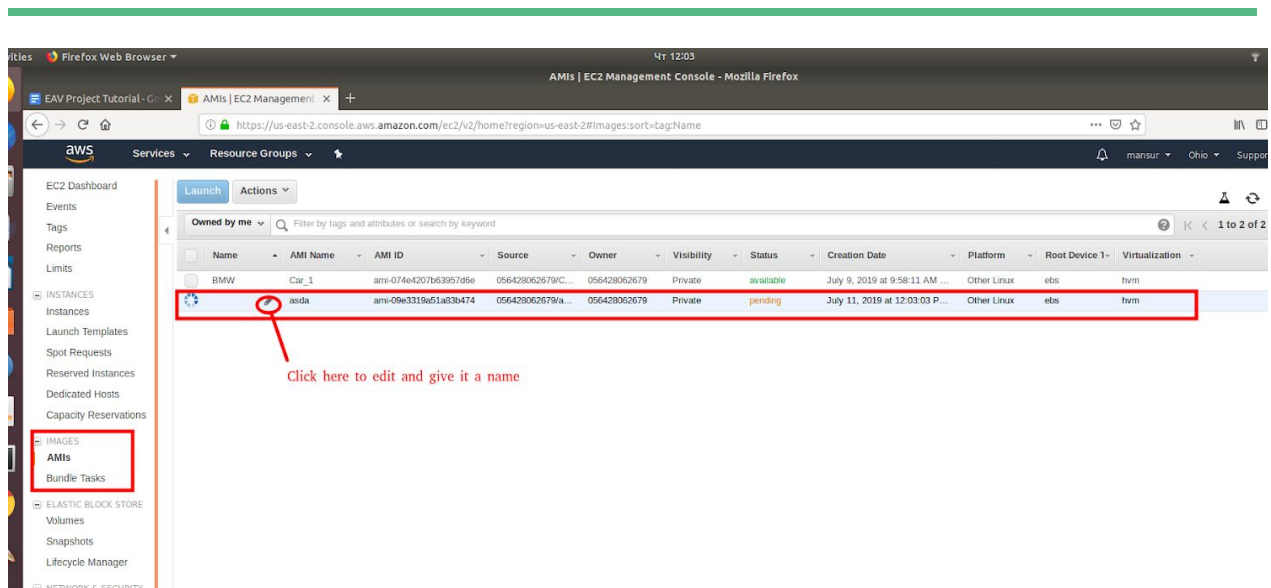
This can be done as follows:

1. Log into AWS console, From *Compute* select *EC2*
2. Then from *Instances* in the left banner select *instances* and select the instance you want to share, and Go to *Actions*
3. Select *Actions-> Image -> Create image*, Give the image a name and a description and select create image

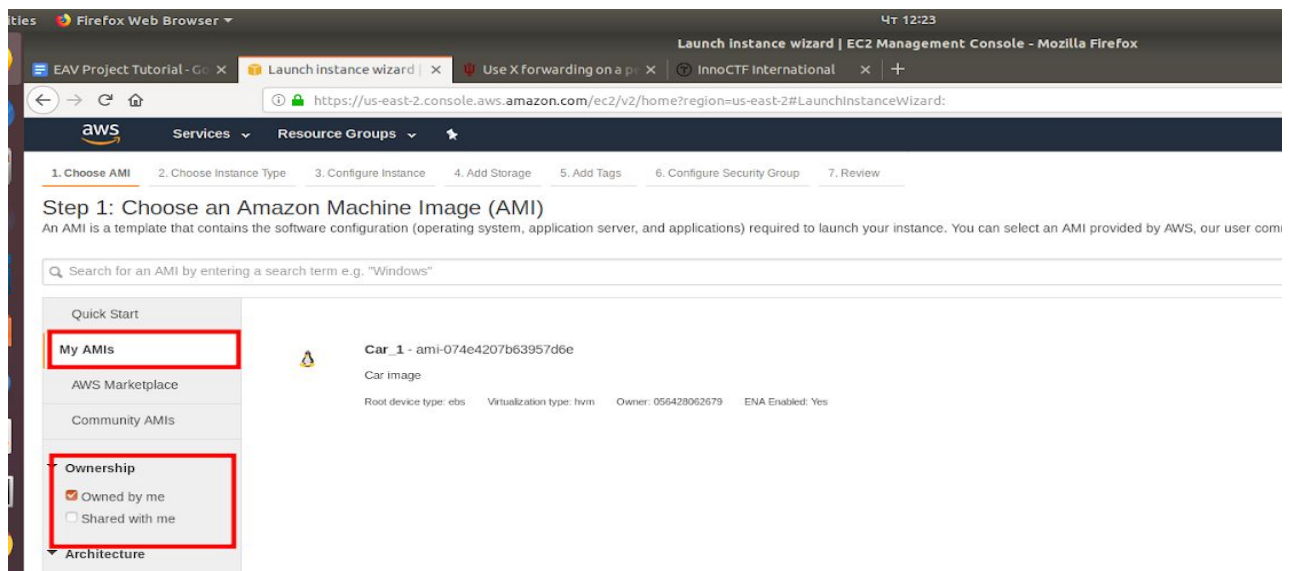


Now we have created an image of the instance we want to share:

4. Under *Images (Left banner)* choose *AMIs*, here you will find the image you created
5. Give this image a name by editing it



6. Select the image and go to Actions -> Modify image permissions
7. You must have the AWS account number that you want to share the image with, write it down and choose *Add permission*
8. Now login from the AWS account that you shared the image with
9. Go to *EC2* -> *Instance* and choose *launch instance*
10. Choose *My AMIs* and select *shared with me* from the left banner



11. You will find the new AMI that you shared, launch the instance and download the key pair to your computer
12. Refer to the previous page to follow the steps of connecting to a machine image

Network communication

To-Do

Charging Experiment

Proof of concept

To prove the feasibility of our proposed scheme, we developed a proof of concept with traditionally available tools such as Raspberry Pi and a temperature sensor (Dallas Semiconductor DS18B20). We consider the case study of EAV charging its battery at a CS or on the move. The temperature sensor is used to indicate that a car is being charged (thus, increasing its temperature)

Architecture

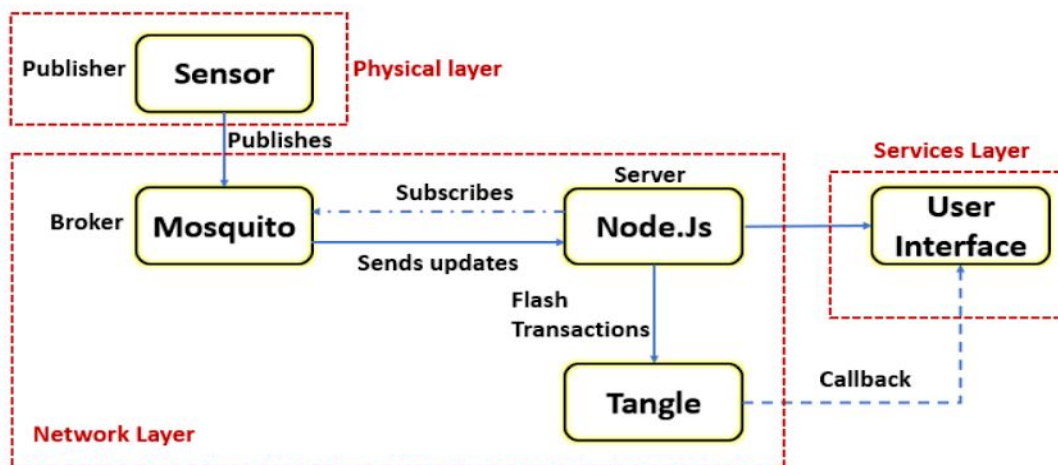
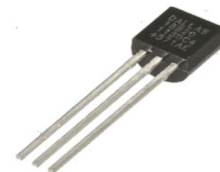


Fig. 1 Communication Architecture



Raspberry Pi 3



Dallas Semiconductor DS18B20
Temperature sensor

Fig. 2 Raspberry-Pi 3 And Dallas Semiconductor

Results

Figure 3. shows the working application in a web browser. Temperature recordings, as well as time stamps, are plotted in the chart in real-time, and the balances of both parties, and the amount of transaction-able IOTAs are shown underneath. This UI is communicating with the Node application through the Web Sockets and HTTPS calls for IOTA Flash transactions.



Fig. 3 Results of the Experiment

Chapter 2. Implementing the decision logic

Summary

Our system consists of two main actors which are: Electric vehicles and Charging stations. Each of which must connect to the other for the process of electricity charging to occur.

In our implementation we supposed that after the vehicle finds the nearest charging station, it will connect to it, knowing that the car was previously registered in that charging station in the given network. So the scenario of the charging process will be as follows:

Scenario steps of the charging process

1. The car searches for the nearest CS (Using Machine learning and Artificial intelligence Algorithms), to find a station in the same network or another nearest one
2. A connection between both the server (CS) and the client (Car) in our process is established
3. After successful connection between both parties, The car sends its **Identification file (.xml file)** to the server (CS)
4. The CS acknowledges the transfer, and approves, or rejects the car depending on certain criteria
5. The CS determines the electricity needed by the car using EVSE smart tool and send it to the car
6. The car accepts or rejects the amount sent
7. If the car accepted the amount, A secure way of payment is established between the car and the CS where the car send the money and the CS send the required amount of Electricity by IOTA

Where each of these steps will be explained in further details.

-
1. The car searches for the nearest CS (Using Machine learning and Artificial intelligence Algorithms), to find a station in the same network or another nearest one
 2. A connection between both the server and the client is established:

In our process, the car represents a client class and the CS represents a server class in Java.

Socket programming with Java:

- A server socket is created for the server(CS) in the port number we want to listen to. The port number is pre-defined between the (CS) and the car or is a default value.
 - For the client (car), a socket is created with the ipnet-address of the CS, and the port number
3. After successful connection between both parties, The car sends its Identification file (.xml file) to the server (CS)

When a user purchases a car, its assigned an identification file which has all the necessary information needed about the car, which is necessary for the process to be completed.

The format of the file is XML to ease the database transactions (MySQL) and it contains the important information about the car:

- a. Car ID (Unique for every car)
 - b. Model name, and year
 - c. Owner's Name
 - d. Owner's phone number
 - e. Owner's ID (Unique for every user)
 - f. Date purchased
-
4. The CS acknowledges the transfer, and approves, or rejects the car depending on certain criteria

This file is sent to the CS to approve (or reject) the car according to a **certain criteria**

The criteria is:

That the car must be previously registered in the CS that it's trying to get charged from, which the user can easily do using the website (Enter website here), We check whether the car ID is previously registered or not.

5. The CS determines the electricity needed by the car using EVSE smart tool and send it to the car
6. The car accepts or rejects the amount sent
7. If the car accepted the amount, A secure way of payment is established between the car and the CS where the car sends the money and the CS sends the required amount of Electricity through IOTA tangle. If it doesn't, the transaction ends here.

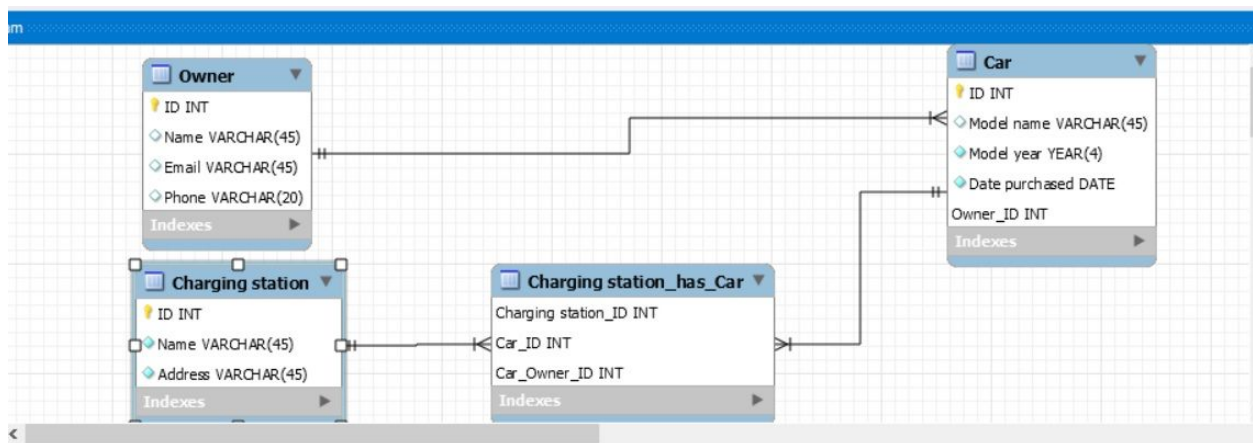
System Database

Entity-Relationship Diagram

The Entity-relationship diagram describing the system will be composed of three main actors:

- Cars
- Charging Systems
- Car owners

A car has definitely only one owner, while an owner can have many cars (**One to many relationship**). A charging station has many cars, and any car can be assigned to many charging stations (**Many-to-many relationship**), which yields a new Relation 'Car_has_CS'



Web application

Car owner registration

Our web application requires car owners to register their information to the database and assign themselves to one/ or more charging stations, this is done either by uploading the xml file/ or by form filling.

Charging station selection

Car owners can assign their cars to one or more stations, to be able to electrically charge from these stations in the future.

Project repository

Link of the project repository

The link to the project repository is

<https://github.com/Ziyadelbanna/Electric-charging-framework>. To use it, you must:

- 1. Clone the project to your local host workspace*
- 2. Open Website from your localhost workspace path*

Important remarks to consider

1. *Java application must be installed in the embedded system of the electric car as well as the charging system, As the very first connection is implemented in Java*
 - *Apache library, and JSON must be considered with the java libraries*
2. *The car owner must register his cars to the charging system prior to charging*
 - *Every car has a unique ID, as well as every person has a unique ID. Also each charging station is identified by a unique ID*