

# IOT Lab 2: IoT Networking

**DISCLAIMER:** *This lab looks long - do not worry! Most of this document is more like reading material than a lab. Our goal is to provide you with extensive descriptions, to walk you through each step, so you will not get stuck, and to also teach you additional things. It is ok to skip over parts of the reading if you feel comfortable and don't feel like you need the extra help.*

## Overview

In the previous lab you gained some experience working with hardware to develop a self-driving car. Good for you! You should now have a basic sense of how IoT devices are constructed, and some level of comfort in working with hardware and software to construct your own device.

But IoT is more than just building individual devices. IoT is about an intelligence. It is about devices working together, sharing information, and merging together with an enormous artificial brain that we are growing in the cloud, which uses devices to watch us, interact with us, and be part of our daily lives to help us do better, teach us, and guide us.

In order for this to happen, devices need to communicate. Luckily, computer scientists have extensively studied how to make IoT devices communicate. We know how to transmit information using a variety of physical media, and not just that, we know how to do that extremely efficiently, both in terms of paradigms that improve robustness and speed of communication, but also in terms of software stacks that simplify application development across the highly complex problem of internetworking communicating devices in the challenging and dynamic environments that IoT devices commonly encounter.

In this lab, you will explore IoT networking by extending your self-driving car with networking capabilities. This is not an artificial exercise in any sort - modern

autonomous vehicles make heavy use of networking infrastructures, enabling not only firmware updates and real-time traffic data, but also things like allowing them to "see around corners" to avoid crashes, coordinate fleet-wide operations, and allow cloud intelligences to diagnose engine faults which require both rapid communication and also multiple networking technologies to work together in a seamless way.

More specifically, in this lab, you will be using two protocols widely used in IoT (Bluetooth and Wifi) to enable communication between your car and the outside world. In particular, you will configure Bluetooth on your Raspberry Pi and Bluetooth module on your laptop to control the movement of your car and display data from your sensors.

## What to Purchase

In order to complete this lab, you will need to purchase some things. Working with real components and assembling them yourself is both important and necessary to give you a more complete, hands-on experience. Since you purchase these components yourself, they are yours to keep, and the thing we construct will be something that you can keep and continue to use (show your friends!). Moreover, many of these components are general-purpose widely used to build IoT devices - if you like, you can later reassemble them into many other amazing IoT projects.

However -- if you did Lab 1, you may already have all the components you need! We will be programming Bluetooth on your laptop/computer -- hence it is important your laptop/computer supports Bluetooth. If it does, you're good to go!<sup>1</sup> If it does not, then you will need to purchase a Bluetooth adapter for your PC. Any USB Bluetooth NIC with Bluetooth version 4.0 or later should work. Here's an example of one that should be safe to purchase:

- *If your PC doesn't have Bluetooth built-in, you will need to get a USB adapter for your PC (Bluetooth 5.0 Support is recommended but not required) ([Link](#))*

## Background

When IoT engineers design systems with networking capabilities, they rarely implement the network from scratch. Networks are complicated enough where you want to draw

---

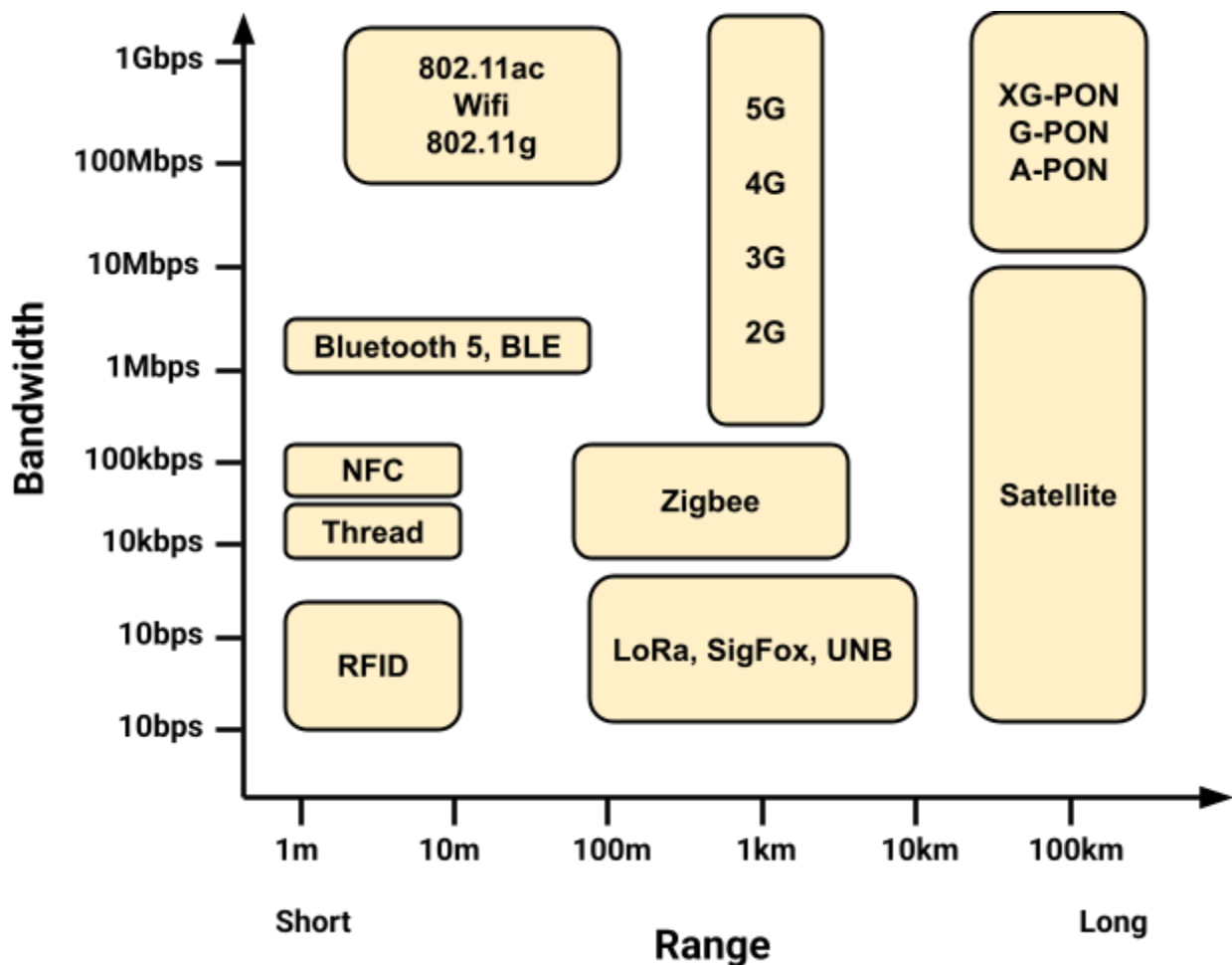
<sup>1</sup> We are assuming your PC has Wifi built in, and that you have a Wifi connection to the Internet. If not, you'll need to get a Wifi NIC for your PC, and perform part of this lab in a location with Wifi Internet access.

upon previous work – there are certain ways that are good to do things, and certain ways that are not so good. Good approaches for doing networking have been codified in the form of "protocols"<sup>2</sup>, and there are certain protocols that have been approved by standards bodies such as the IEEE, ITU, TIA, etc. When you design systems it is smart to leverage existing wireless protocols rather than designing something from scratch – in fact, it is common to not only leverage existing protocols (which have been written up in standards documents, which you can follow to realize your own implementation), but to directly leverage hardware implementations of those protocols, e.g., instead of implementing 802.11ax yourself you'll shop around and purchase a bunch of 802.11ax chips from a particular vendor, study their API, and then program directly to that API.

Given that, the question IoT engineers then often grapple with is, which protocol should I use? There are many wireless protocols out there that have been used for IoT devices: RFID, 6lowPAN, Bluetooth, Bluetooth Low Energy (BLE), Z-Wave, 802.11ah (HaLow), SigFox, LoRaWAN, EC-EGPRS, GPRS, Zigbee, Thread, NFC, NB-IOT, 4G/5G, Ingenu, Weightless-N, MiWi, LPWAN, etc. How to choose between them seems daunting until you realize there is a fairly clear taxonomic procedure you can follow to do that. Protocols differ in terms of their carrier frequency—which affects how far the signals propagate, how well they propagate through the atmosphere and obstacles—and transmission power—which affects how far the signals propagate and how much energy they require as well as data rate. In general, these protocols can be grouped into several categories based on their typical target use case, as shown in the figure below:

---

<sup>2</sup> "Protocols" are programmatic structures that dictate how various participants in a network can communicate - they define things like how to translate data into bit sequences, what order data items should be transmitted in, what messages can be sent back and forth and such. In addition to protocols, networking engineers also think a lot about "architectures" (approaches for layering/combining protocols/devices, for example particular topologies or the OSI layering model) and "policies" (goals the operator may have for the network, which may be realized through manipulating parameters or customizing configurations of protocols).



**Figure 1: IoT protocols differ in their bandwidth (speed they communicate at) and range (how far their signals propagate). When designing an IoT system, use this information to choose the IoT protocol best for your requirements.**

A related question is how you deploy your network - you could deploy your own, and for a lot of kind of shorter-range deployments this makes sense. But if you've got a large deployment stretching over a long range you may want to use an existing provider. For example many cellular providers have entered the IoT space, offering traditional cellular data plans (e.g., 5G/4G LTE) as well as protocols like NB-IoT which are targeted at lower bandwidth (yet long range) applications. The benefit of choosing an existing provider is you don't have to go out and deploy your own infrastructure (eg if you're working for the government and doing some traffic monitoring in a metropolitan area, you probably don't have the budget to deploy your own cellular towers or satellite infrastructure, and you really don't need to given there are often going to be multiple providers with infrastructure like that already providing coverage for your deployment area).

Ok so if you were Tesla and you wanted to deploy networking for your cars, what might you do? Well, first you might want to think about the use cases for networking on your devices - perhaps you want some way to disseminate firmware updates, some way to monitor where your cars are located, receive camera feeds to teach your computer vision algorithms, communicate between your cars to avoid crashes, speak with the customer's cell phone to allow them to start the car, some sort of near-field communication to read a card held up to the door to unlock it, etc etc. If you think about these cases, you may realize multiple modes of networking may make sense here - perhaps you'd use RFID to read cards, BLE to communicate with the user's phone, Satellite or 4G to receive updates from cars, etc. Perhaps you'd use some custom protocol for crash avoidance since there's not really an existing protocol standardized for that sort of thing yet.

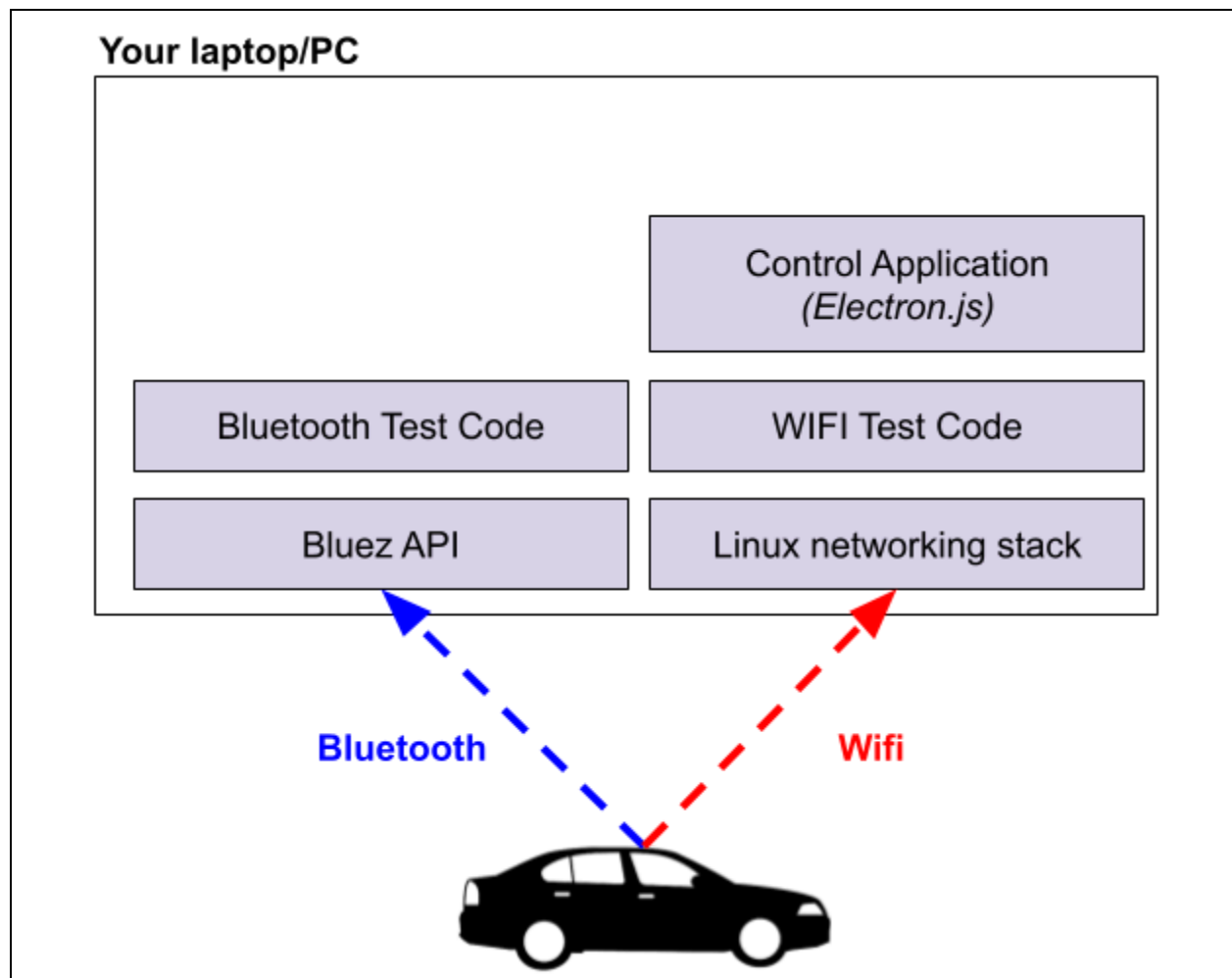
For this assignment, we want you to think about what wireless technologies you would ideally use. If you were designing a real, self-driving car platform, with real (big) cars, that would drive all over the United States, which technology would you use? **Please include in your report a description and analysis of what protocols you would use and why (brief is fine - a paragraph or so).**

However, to keep things kind of standardized, and to ensure you get a bit of breadth in your experience, we'll tell you what to implement. You are going to be working with Bluetooth and Wifi in this assignment. As you go through these two protocols you'll notice commonalities in their APIs - please be aware that while there are many other protocols out there, the way they're programmed is actually quite similar to how these two protocols are, so the experience you'll gain in this lab will have some good generality.

Here's a bit of background on those protocols. Both Bluetooth and Wifi are two standards of wireless communication. You may have heard of these protocols before. Bluetooth has been historically used for transferring small data between personal devices (phones, laptops, headphones, mice etc.) when speed is not the top priority (a few megabits per second). On the other hand, WIFI is able to transfer data much faster (few hundred megabits per second) with a substantially larger range. These protocols were designed for different use cases: you may only want to use your wireless headphone when your phone or laptop is near you whereas you can access campus Wifi anywhere on a college campus. So if Wifi is so much better, why does Bluetooth exist? Well the reason is that Bluetooth has some benefits of its own: it's designed to use less power, hardware can be cheaper. When we build circuits, it turns out it's cheaper to build

circuits that use lower frequencies (electricity oscillates back and forth in them slower), whereas higher frequency/bandwidth circuits are more expensive. Same for battery usage, when you transmit at higher frequencies you have to clock your circuit higher, you have to use smaller components, you use more electricity. It's the same reason why CPUs get more expensive and use more power the faster they get. These same trade-offs exist for all the other wireless protocols as well. So it really depends on your needs. Since Bluetooth only handles small data in small ranges, it consumes much less power. Many wireless mice and keyboards can be used for weeks without charging.

In this lab, you'll be using both protocols however, as different parts of your car have different needs. You'll be using Bluetooth for local control functions - to display basic information about your car, and to start and drive it. You'll be using Wifi to communicate with a simple cloud backend (which you'll extend in Lab 3 to perform analytics on your car). We'll be using Bluetooth to give you experience with peer-to-peer communications, with the car communicating directly with your laptop, and Wifi to give you experience with communicating via the public Internet, to a REST endpoint in the cloud. The overall design of what you'll create is shown in the following figure:



**Figure 2: System architecture. The variant of 802.11 that will be used will be automatically determined<sup>3</sup> based on the variant supported by your home's access point and the variant supported on your car.**

This lab consists of four parts. First, we will set up Bluetooth networking on your car, and perform some mobile Bluetooth programming to create data structures and export them over wireless. Second, we will perform some Bluetooth programming on your

<sup>3</sup> This process happens automatically via a negotiation process on the first connection - it's part of the protocol, nothing you have to deal with in your code. The variant of wifi that will be used is determined by a selection procedure that prefers more recent/appropriate wifi versions. As of this writing the current version of the Wifi protocol is 802.11ax (Wi-Fi 6). Previous to that there was 802.11ac (2014), 802.11n (2009), 802.11g (2003), etc. Newer protocols often have improved physical layer designs (more advanced ways to encode bits into wireless signals), improved security, etc, but are designed to be backwards compatible (imagine what a nightmare it would be to replace all your wireless devices every time a new wifi version came out!)

computer as well, to query and receive data using a Bluetooth API, which we will then display by using Javascript to create a user interface (leveraging Electron.js).

Your implementation will be able to do things like the following when it is completed:

- Stream car data to the PC -- we suggest the car's location, battery level, temperature of the pi, speed of the car, and total distance traveled should be sent (at minimum, please send at least three pieces of information).
- Stream commands from the PC to the car -- in particular, the PC should be able to control the car by changing the steering left and right, and changing the motors to go forward and backwards (for example, you could display up/down/left/right arrows).

## Step 1: Create the Bluetooth Connection (Extra Credit)

So one thing we clearly need to do is to send data using Bluetooth. Luckily, you don't have to implement Bluetooth by yourself - it is such a widely used protocol that most operating systems support it, including Raspberry Pi OS.

The goals of this section are as follows:

- Create a simple frontend mobile application that can communicate with the Raspberry Pi via Bluetooth serial using RFCOMM
- Update existing Python codebase to communicate with the mobile application
- Alternatively, you can use windows/linux pc to implement the bluetooth communication via sockets (Section 4)

Note: For the mobile approach, you will need a physical Android device. iOS devices will not work. Emulator will not work.

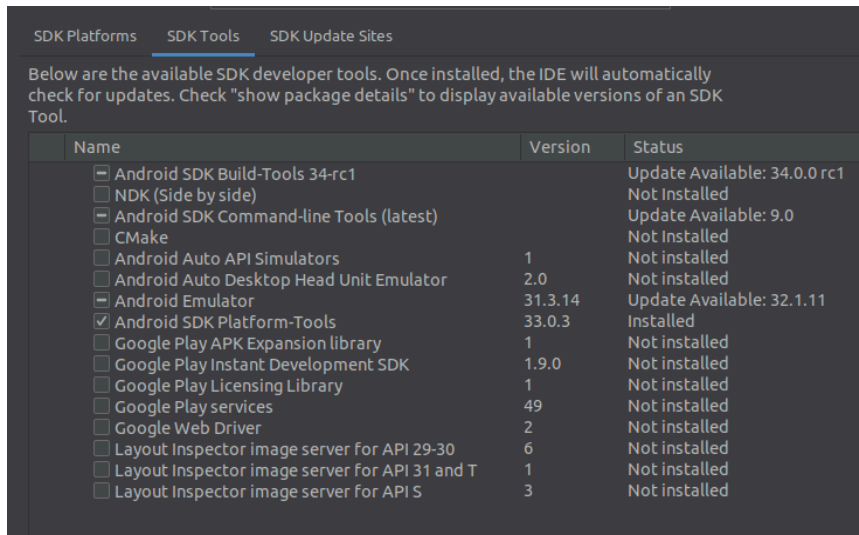
Setup PC:

- Install Flutter from <https://docs.flutter.dev/get-started/install> . There are instructions for every major OS including M1 MacOS. Make sure to also set up your code editor (VSCode).
- Check your flutter installation with "flutter doctor" and fix any missing requirements. Keep in mind that we only need Android Studio. Also make sure the flutter bin is in your path.



- Note: For this step, it is ideal to select the OS version that your Android device is running.

Once Android Studio is installed, proceed to open the application. Click on “More Actions” -> “SDK Manager”. Then, under the sdk platform, checkmark “Android 12.0(S)”. Under “SDK Tools”, checkmark “Android SDK Platform-Tools”. Click on Apply and wait for installation



- At this point there should not be any more errors when running “flutter doctor” in the terminal.
- Next, follow the instructions here to setup your Android device <https://docs.flutter.dev/get-started/install/linux#set-up-your-android-device> . Running “flutter devices” while your Android devices is connected to your PC should display device name and OS version

#### Setup Raspberry Pi:

- Click on the Raspberry Pi icon -> Preferences -> Raspberry Pi Configuration. On popup, make sure to toggle on “Serial Port” and “Serial Console”. Then restart your Pi.
- Now we need to edit the “bluez.service” configuration file. Open a terminal and type in “sudo nano /etc/systemd/system/dbus-org.bluez.service”. This should display a file in the terminal. You want to add “-C” at the end of the line starting with “ExecStart”. Also add the line “ExecStartPost=/usr/bin/sdptool add SP”. The

resulting file should look like this:

```
GNU nano 6.2 /etc/systemd/system/dbus-org.bluez.service *
[Unit]
Description=Bluetooth service
Documentation=man:bluetoothd(8)
ConditionPathIsDirectory=/sys/class/bluetooth

[Service]
Type=dbus
BusName=org.bluez
ExecStart=/usr/lib/bluetooth/bluetoothd -C
ExecStartPost=/usr/bin/sdptool add SP
NotifyAccess=main
#WatchdogSec=10
Restart=on-failure
CapabilityBoundingSet=CAP_NET_ADMIN CAP_NET_BIND_SERVICE
LimitNPROC=1

# Filesystem lockdown
ProtectHome=true
ProtectSystem=full
PrivateTmp=true

^G Help      ^O Write Out ^W Where Is  ^K Cut       ^T Execute  ^C Location
^X Exit      ^R Read File ^\ Replace   ^U Paste     ^J Justify  ^_ Go To Line
```

- Save and exit the file. Then run the following commands in the terminal:
  - “sudo systemctl daemon-reload”
  - “sudo systemctl restart bluetooth.service”
- We will also install a library called “bluedot” by running “sudo pip install bluedot”
- Finally, you should pair your mobile device with the Raspberry Pi either through the terminal or using the GUI (hit the top right bluetooth icon). The terminal method is outlined here:

<https://bluedot.readthedocs.io/en/latest/pairpipi.html#using-the-command-line>

It is highly recommended to work with the provided flutter starter code. We will walk you through the example which should give you a better sense of how to work with the bluetooth library. However, first, it might be beneficial to go through a simple example of a flutter project:

<https://codelabs.developers.google.com/codelabs/flutter-codelab-first#0> .

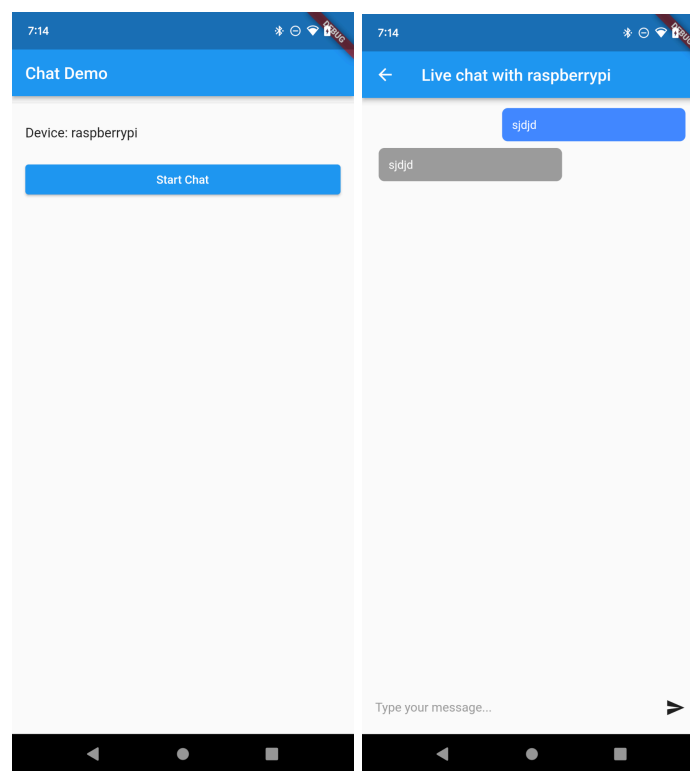
## Section 1: Mobile App

First clone the following repository: <https://github.com/Saurabhdarekar/Flutter-app> .

After opening the project in VSCode, the editor should prompt you with running pub get which will download all the dependencies. On the PC side, your work should be done in the dart files. If you want a better grasp of the dart language, you can look at

<https://dart.dev/> .

The example application given is a chat application. The user starts at the main page, and can then click on “chat” to begin chatting. In terms of actual bluetooth portion, the first screen will acquire the bluetooth device object, while upon entering the second screen a connection will be established between your device and the pi. Currently, the device is identified as the pi by the advertised name “raspberrypi”. Also, we are only checking bonded devices so it is important to pair the PC and Pi beforehand. The device reference gets passed from the first screen to the second. You will work on replacing the second screen with relevant UI for your project. The starting skeleton code is also given (“AppPage.dart”) for you to work on. In “MainPage.dart” on line 61 replace “\_startChat(…” with “\_startApp(…” so that the button leads to the new page.



The order of the screens (left to right): main screen, chat

Let’s try to compile or build the mobile app. Run the command “flutter build apk” to build your android app. Depending on the SDK you selected there might be some issue with the gradle files. The build command will inform you if there are issues that need to be resolved. Once the problems are resolved and the build succeeded, we can pause and temporarily move to the raspberry pi code.

Note: In order for the live chat to work properly, you will need to setup the server in the next section

## Section 2: Raspberry Pi

To keep things simple we will be using the bluedot library to establish the bluetooth server. Specifically, we will be working with the comms API <https://bluedot.readthedocs.io/en/latest/btcommapi.html> . Below is the code used on the Pi.

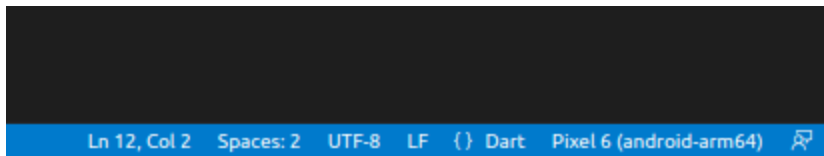
```
1 from bluedot.btcomm import BluetoothServer
2 from signal import pause
3
4 def received_handler(data):
5     print(data)
6     s.send(data)
7
8 s = BluetoothServer(received_handler)
9
10 pause()
```

This snippet of code will sleep until data is received. Then, it prints the data and sends it back to the sender. There is nothing really fancy going on here. However, keep in mind that the project will require you to send data from the phone to the Pi (button press, commands etc) while also receiving periodic status updates.

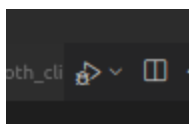
## Section 3: Tying it all together

To run and debug your application, you should first make sure that bluetoothserver is running on the RaspberryPi. Then you will want to run/debug the mobile app. In the past, you had to solely rely on the CLI. Now, with VSCode integration, it is much easier.

- You need to first select the device to run the application on. This can be selected on the bottom right corner of VSCode}



- Click the arrow and gear icon top right of the editor to begin debugging



## Section 4: Create the Bluetooth Connection without Mobile App

We can also implement bluetooth without a mobile app! This is an alternative for those without Android devices. However, keep in mind that bluetooth code might be problematic to run within a linux VM. Therefore, if possible, it is recommended to write and run your bluetooth application on native OS (Windows, Linux).

Note: Your system might have overwritten the socket.py library from standard Python. Personally, I had problems with Anaconda messing with the library which prevented me from using “AF\_BLUETOOTH”. As of now, the solution is to uninstall Anaconda and reinstall python. It is also important that Windows users install python3 version > 3.9 as this version is where AF\_BLUETOOTH is first supported. To test whether your socket.py library is working correctly, run “windows\_socket.py” from the repository ([https://github.com/rickyhh2/bluetooth\\_rfcomm\\_socket](https://github.com/rickyhh2/bluetooth_rfcomm_socket)). As long as your terminal is not showing errors like “AttributeError: module 'socket' has no attribute 'AF\_BLUETOOTH'”, your python installation should be fine. Use ctrl+c to exit the program.

Setup:

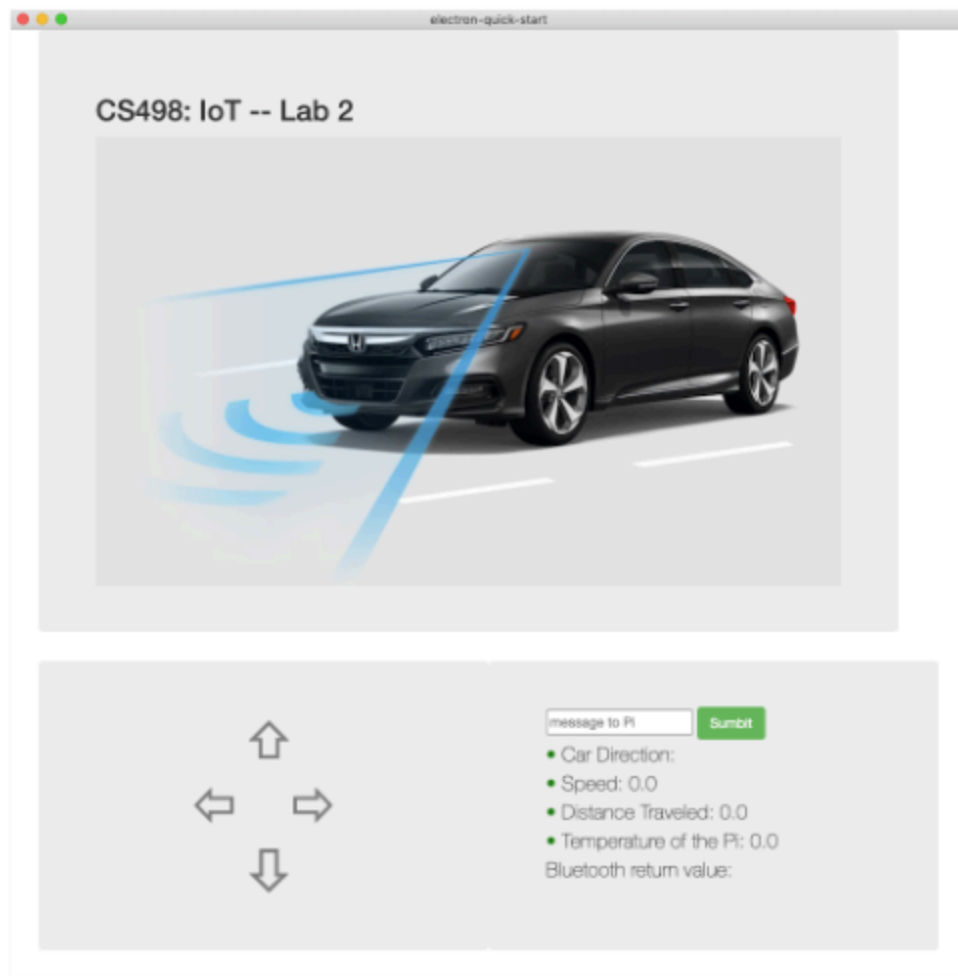
- Follow the setup for Raspberry Pi from step 1 if you haven’t done so already
- Place pi\_socket.py in Raspberry Pi and widnows\_socket.py on your PC
- Replace the server\_addr variable with the MAC address of your RPi.
  - One way is to use bluetoothctl to scan for the MAC address
  - Alternatively, you can use the bluedot mobile app to pair with your device to figure out your MAC address
- Make sure your RPi is paired with your PC.

To run the demo application, first run the python code on RPi as it is the acting socket server. Then, run the python code on your PC. You should see a bunch of print messages like “PC [#] “ or “RPi [#] “. This signifies that you have successfully created some form of communication between the two devices!

If you have coded in python or c for other networking courses before, you might be familiar with socket programming. Here’s a link to a page explaining what socket is ([https://en.wikipedia.org/wiki/Network\\_socket](https://en.wikipedia.org/wiki/Network_socket)). Basically we create a socket connection between the RPi and the PC to support bidirectional serial communication. This way you can utilize the same socket for both reading and writing.

As stated previously in Step 1, you need to not only be able to send information (buttons controls), but also receive information from the RPi. Therefore, it may be useful to use either multithreading or multiprocessing. We highly recommend reading the documentation (<https://docs.python.org/3/library/socket.html>) for a better understanding of each function used.

Regarding the UI, there are a couple options that you can explore. For example, python-shell is something that you can work with to run python code from a node based application like an Electron app (<https://www.npmjs.com/package/python-shell>) which you will implement in the next step. Alternatively, you can use Tkinter (<https://realpython.com/python-gui-tkinter/>) or other UI libraries to build a standalone application just for the bluetooth portion.



(Optional) The car can also stream a real-time view from a PiCam to the PC for display in the Electron UI! It is not required to do this but this is something your platform would

be fully capable of doing if you chose to add this functionality (and might make your car more fun, so you can drive it into other rooms while you stay in another room).

## Step 2: Create a Web Service

In this step, you will set up a simple web app using Python and JavaScript/HTML and connect it to your car using Wifi. You should achieve and expand the functionalities in step 1 with WiFi, and display necessary information in the web app. If you like, you can just download the code for this step at

[https://github.com/mccaesar/iot-labs/tree/master/iot-lab-2/frontend\\_tutorial](https://github.com/mccaesar/iot-labs/tree/master/iot-lab-2/frontend_tutorial) .

However, for educational purposes, this step will walk you through all the steps of creating that code. We ask you to please go through this walk through, so you fully understand the steps.

### Pre-requisites:

1. Make sure you already have python3 installed on your computer (<https://www.python.org/downloads/>) .
2. Make sure you already have ElectronJS and also NodeJS installed. To get NodeJS working, download and install the files at <https://nodejs.org/en/download/>. Read the ElectronJS tutorial link (<https://www.electronjs.org/docs/tutorial/quick-start>) and use the following steps to setup an Electron JS project work directory:
  - Create a new folder for this this project
  - Inside the new folder, initialize this folder as an Electron project work directory. You need to put all of your files in the same folder for the following steps because Electron will only look for the files in its work directory.
    - \$ npm init -y
  - Download Electron packages
    - \$ npm install electron

## Section 1: WIFI connection

Using WIFI to communicate with your car is quite similar to the process of configuring Bluetooth in the last step. Again, you can feel free to use the code we provided as a

starting point

([https://github.com/mccaesar/iot-labs/tree/master/iot-lab-2/frontend\\_tutorial](https://github.com/mccaesar/iot-labs/tree/master/iot-lab-2/frontend_tutorial)).

When you perform these steps, it is ok to do them right on your laptop/computer. If you like, it is also ok to set up a virtual machine to do them in. Doing these steps in a virtual machine is not necessary, but we do recommend you work in the same environment as the one you worked in with bluetooth connection to prevent the need to move things back and forth between environments.

1. Make sure you have connected your raspberry to Wifi (you probably already did this in lab 1 -- if you did not, take a look back at lab 1 to see how to do this).
2. Find the IP address of your Raspberry PI by typing the following command line in your PI terminal.

o \$ ifconfig

```
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.81.128 netmask 255.255.255.0 broadcast 192.168.81.255
    inet6 fe80::3a06:edad:5644:533c prefixlen 64 scopeid 0x20<link>
    ether 00:0c:29:df:e6:76 txqueuelen 1000 (Ethernet)
    RX packets 734165 bytes 764128827 (764.1 MB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 182954 bytes 22702177 (22.7 MB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 2648 bytes 253350 (253.3 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 2648 bytes 253350 (253.3 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

- o You should see something like the above. This command lists the interfaces (network connection points) on the Raspberry Pi. If you have multiple network interfaces configured, you'll see them all listed. They have different names -- "lo" is the "loopback interface", that's just a virtual interface the OS can use to send data to itself, you can ignore that one. You want to look for the wireless interface, which should say "Ethernet" somewhere in it. To figure out the IP address of your Wifi interface, look over the output to find the wifi interface, and look for the "inet" field -- the IP address will follow that. An example is shown above, see the red box.
3. You can check whether the Pi is reachable to you by pinging the Raspberry Pi's IP address in your PC's terminal.<sup>4</sup>

o \$ ping <Pi IP Address>

```
PING 192.168.81.128 (192.168.81.128): 56 data bytes
64 bytes from 192.168.81.128: icmp_seq=0 ttl=64 time=0.328 ms
64 bytes from 192.168.81.128: icmp_seq=1 ttl=64 time=0.511 ms
64 bytes from 192.168.81.128: icmp_seq=2 ttl=64 time=0.504 ms
64 bytes from 192.168.81.128: icmp_seq=3 ttl=64 time=0.509 ms
64 bytes from 192.168.81.128: icmp_seq=4 ttl=64 time=0.478 ms
64 bytes from 192.168.81.128: icmp_seq=5 ttl=64 time=0.519 ms
64 bytes from 192.168.81.128: icmp_seq=6 ttl=64 time=0.489 ms
64 bytes from 192.168.81.128: icmp_seq=7 ttl=64 time=0.521 ms
64 bytes from 192.168.81.128: icmp_seq=8 ttl=64 time=0.499 ms
^C
--- 192.168.81.128 ping statistics ---
 9 packets transmitted, 9 packets received, 0.0% packet loss
```

<sup>4</sup> Having trouble w  
status). If you do  
number)). You can  
(telnet [ip ad



- o The "ping" utility sends a small packet to an IP address, which reflects it back. The ping utility will print out information to indicate what's happening to these probe packets, how long they take to get back, and whether any get lost and such. In the example above, we're getting replies back, meaning the Pi is reachable. Ping will happily keep sending probes forever, to stop it, hit control+C.
  - o If you get pings back, then you know your computer can talk over Wifi to the Raspberry Pi.
4. Write down the IP address of your Raspberry Pi's Wifi interface that you discovered above. You will need it in the following steps.
  5. Run the provided code `wifi_server.py` on your Raspberry Pi and `wifi_client.py` on your PC. Before you do that, please change the IP addresses in both files to the address you found in the last step. Just like in the Bluetooth code, `wifi_client.py` on your PC will ask for your input message and send it to the Raspberry Pi server, and the server `wifi_client.py` will simply echo back the message to the client. Think about how to expand this functionality in the provided code to transmit necessary information to/from the car (for example, you send the message "forward" and the car will move forward for 1 meter. ). We will explore how to embed this into the web app in the next few sections.

## Section 2: Render the HTML Page with Electron JS

1. In the electron work directory, create an HTML file and name it "index.html".
2. Write the following HTML code in the index.html file. This will be the content for your frontend page.

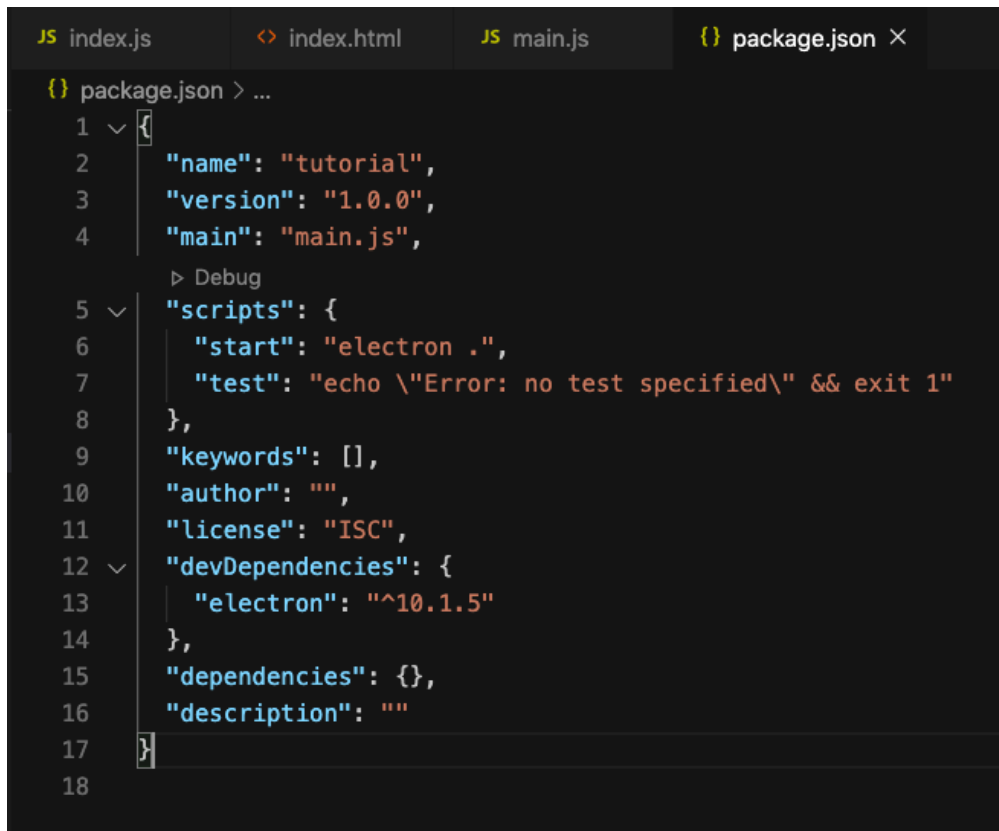
A screenshot of a code editor with a dark background. The editor has two tabs at the top: 'app.py' and 'index.html'. The 'index.html' tab is active, showing the following HTML code:

```
<? index.html > ...
1  <html>
2
3  <body>
4  <h1>My First Website</h1>
5  <p>Hello IoT!</p>
6
7  </body>
8
9  </html>
```

The default port number is 65432.

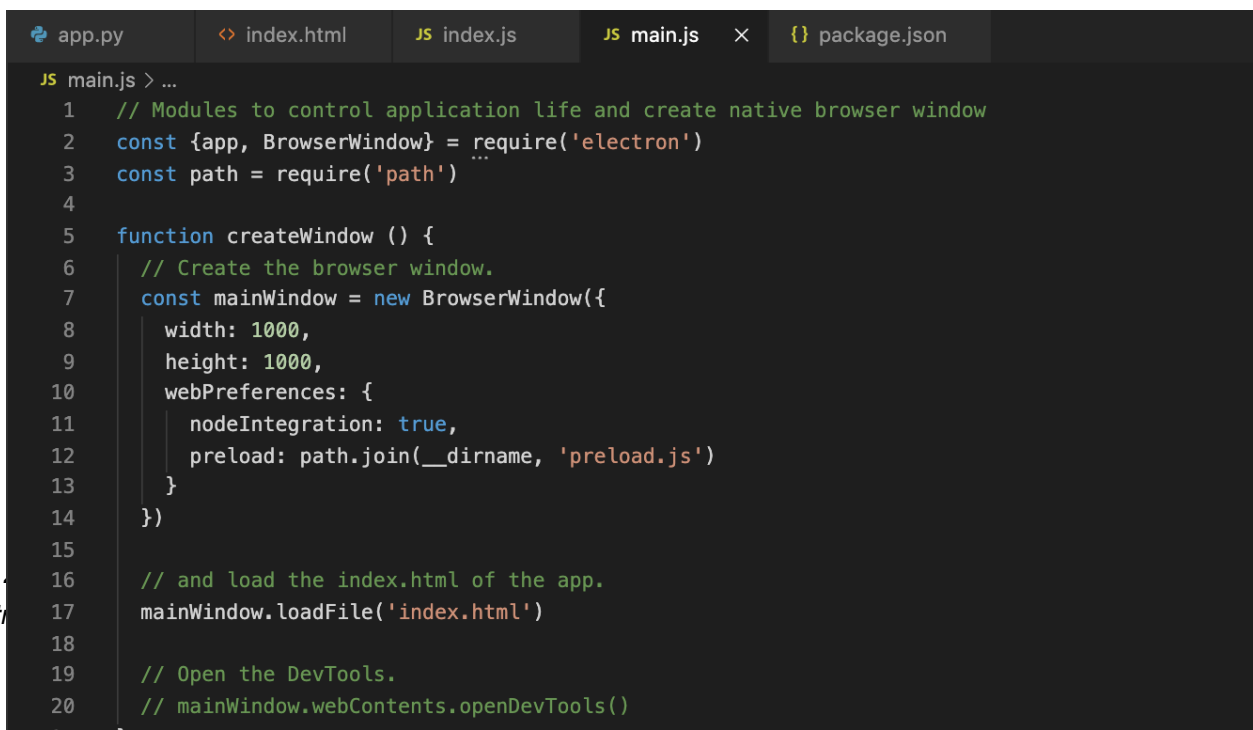
To have Electron render the index.html file, we need to do a few more things:

3. You should be able to see a package.json file in your directory. Update it as follows (make sure line 4 is "main.js", and add line 6 to the file as follows):



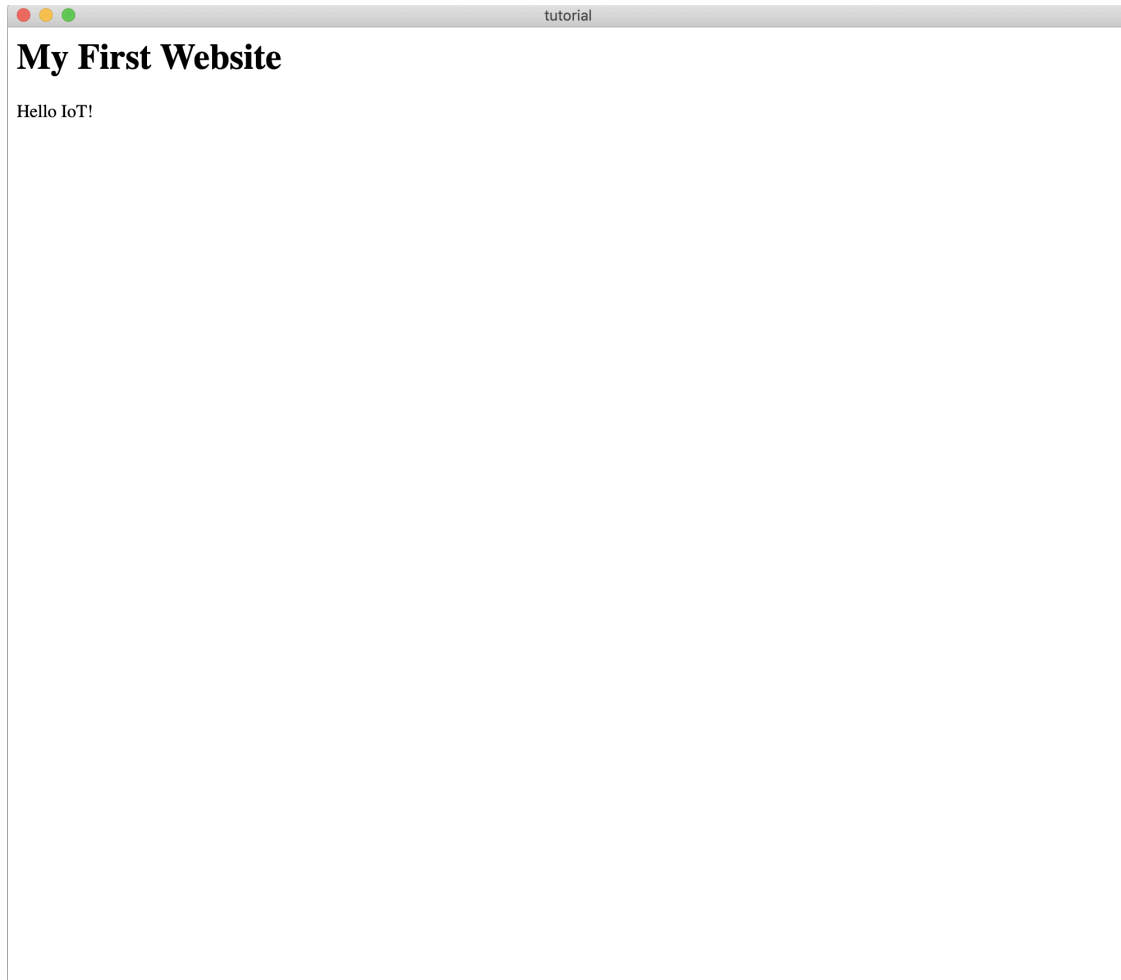
```
{"name": "tutorial",  
  "version": "1.0.0",  
  "main": "main.js",  
  > Debug  
  "scripts": {  
    "start": "electron .",  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "electron": "^10.1.5"  
  },  
  "dependencies": {},  
  "description": ""  
}
```

4. Create another files called "main.js", and type in the following program:



```
// Modules to control application life and create native browser window  
const {app, BrowserWindow} = require('electron')  
const path = require('path')  
  
function createWindow () {  
  // Create the browser window.  
  const mainWindow = new BrowserWindow({  
    width: 1000,  
    height: 1000,  
    webPreferences: {  
      nodeIntegration: true,  
      preload: path.join(__dirname, 'preload.js')  
    }  
  })  
  
  // and load the index.html of the app.  
  mainWindow.loadFile('index.html')  
  
  // Open the DevTools.  
  // mainWindow.webContents.openDevTools()  
}
```

5. Now, if you run `$ npm start`, you should be able to launch the web app using Electron. When you do that, you should see output like this:

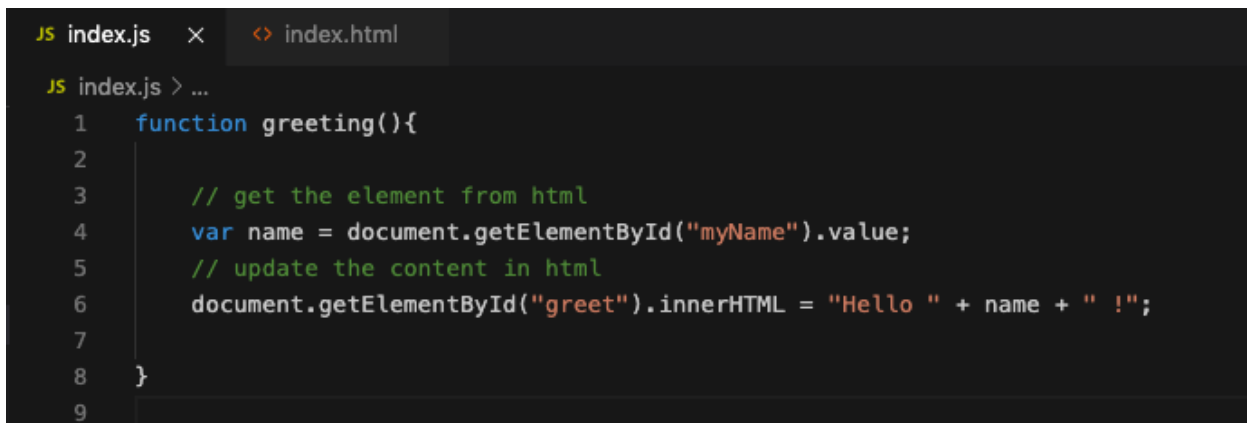


You have successfully rendered a web app using Electron JS! The command `$ npm start` will look for the `package.json` file in your work directory. This file contains the configurations of the Electron web app. For instance, line 4 specifies the “main” function that instantiates the app. The `main.js` file contains the actual setting for your web app. For example, the height and width of the window at line 8 and 9, and the home page at line 17. If you are interested in more details, please refer to: <https://www.electronjs.org/docs/tutorial/quick-start> .

## Section 3: Extend the JavaScript Code

JavaScript is a powerful platform and we've only scratched the surface of what it can do. In this section, we'll go a bit further, by writing more JavaScript to add a few more functionalities.

1. Create a JavaScript file and name it `index.js`. Enter the following code in your `index.js` file:
  - Line 4 get the value of the element "myName" from html
  - Line 7 updates the value of the element "greet" in html

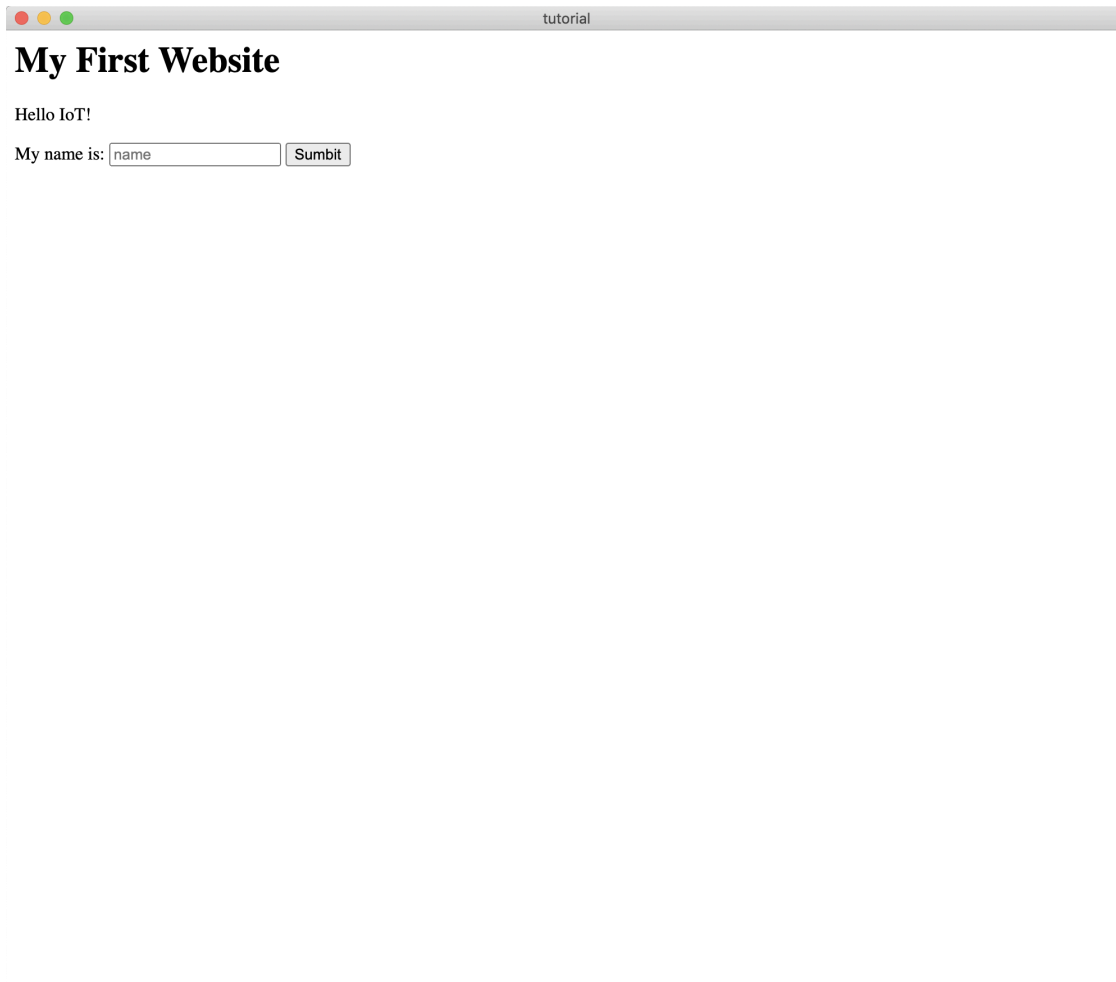
A screenshot of a code editor with two tabs: 'index.js' and 'index.html'. The 'index.js' tab is active, showing a JavaScript function 'greeting()' with the following code:

```
1 function greeting(){
2
3     // get the element from html
4     var name = document.getElementById("myName").value;
5     // update the content in html
6     document.getElementById("greet").innerHTML = "Hello " + name + " !";
7
8 }
9
```

2. Update the `index.html` file as shown in the figure below:

```
app.py  index.html x
<> index.html > ...
1  <html>
2
3  <script src="index.js"></script>
4
5  <body>
6  <h1>My First Website</h1>
7  <p>Hello IoT!</p>
8
9  My name is: <input id="myName" type="text" placeholder="name"/>
10 <button class="btn btn-success" onclick="greeting()">Submit</button>
11 <br>
12 <span id="greet"> </span>
13
14 </body>
15
16 </html>
```

- Line 3 imports the javascript file to the html.
  - Line 9 creates an input textbox with id "myName"
  - Line 10 creates a button. It will call the function greeting() in index.js when clicked.
  - Line 12 creates an empty element with id "greet"
3. Then, launch the web app again by running `$ npm start`. You should be able to see this:



- Write something to the textbox and click on “Submit”. What do you see? Do you see what you expected? Play around with the form a bit and see if you can understand what your code is doing, and how it results in the output you see.

## *Section 4: Communication between frontend and backend*

In the previous sections, we made the Raspberry Pi and your PC talk to each other using Wifi, and we designed a simple frontend web app. In this section, we will combine them together. We will still have the Raspberry Pi act as a server that can serve up web pages and data, but this time, we will let the web app talk to the server and display the response on the web app. Let’s get to it! Please perform the following steps:

1. Update the index.html file as follows:

```
app.py  index.html x
<> index.html > ...
1  <html>
2
3  <script src="index.js"></script>
4
5  <body>
6  <h1>My First Website</h1>
7  <p>Hello IoT!</p>
8
9  My name is: <input id="myName" type="text" placeholder="name"/>
10 <button class="btn btn-success" onclick="greeting()">Submit</button>
11 <br>
12 <span id="greet"> </span>
13 <br>
14 <span id="greet_from_server"> </span>
15 </body>
16
17 </html>
```

- The `<br>` tag creates a new line
  - Line 14 adds a new element with id `"greet_from_server"`. We'll leave the contents of this empty for now.
2. Next, let's write a function to send the user input message to the Raspberry PI server and receive feedback from it. Add the function `client()` as shown below in the `index.js` file, and update the `greeting()` function as follows. The `client()` function will substitute the functionality of `wifi_client.py`. You can refer to code in `iot-lab-2/frontend_tutorial`. (in `index.js`, feel free to remove the line `to_server(name)`.)
- The `client()` function will first connect to the server. Please change the IP address and the port number in the code to be the server address and port number. The code will then get the response from the server and update the value of `"greet_from_server"` on the HTML page.
  - Notice that `client()` will be called at the end of `greeting()` function. What it does is that after clicking on the Submit button, `greeting()` will first update the text on the html page as in the last section, then it will call `client()` to send the input text to the PI server, and get a response from it. Instead of repeatedly asking the user for the input message as in `wifi_client.py`, the message will only be sent when the user client clicks on the "Submit" button.

- Need to add one extra line in the `createWindow()` function in `main.js`:

```
function createWindow () {  
  // Create the browser window.  
  const mainWindow = new BrowserWindow({  
    width: 1000,  
    height: 1000,  
    webPreferences: {  
      nodeIntegration: true, //you might need to  
add this too  
      contextIsolation: false, // Add this  
parameter setting  
      preload: path.join(__dirname, 'preload.js')  
    }  
  })  
}
```



```
<> index.html  JS index.js  X
JS index.js > ...
1  var server_port = 65432;
2  var server_addr = "192.168.3.49";  // the IP address of your Raspberry PI
3
4  function client(){
5
6      const net = require('net');
7      var input = document.getElementById("myName").value;
8
9      const client = net.createConnection({ port: server_port, host: server_addr }, () => {
10         // 'connect' listener.
11         console.log('connected to server!');
12         // send the message
13         client.write(`${input}\r\n`);
14     });
15
16     // get the data from the server
17     client.on('data', (data) => {
18         document.getElementById("greet_from_server").innerHTML = data;
19         console.log(data.toString());
20         client.end();
21         client.destroy();
22     });
23
24     client.on('end', () => {
25         console.log('disconnected from server');
26     });
27
28 }
29
30
31 function greeting(){
32
33     // get the element from html
34     var name = document.getElementById("myName").value;
35     // update the content in html
36     document.getElementById("greet").innerHTML = "Hello " + name + " !";
37     // send the data to the server
38     to_server(name);
39     client();
40
41 }
```

3. Run the `wifi_server.py` code on the Raspberry Pi and start the Electron app with the command `$npm start`. Type something in the textbox in the web app and click on the "Submit" button. Do you see the output you expected? Notice that the Raspberry Pi server code will print the received messages' client IP address in the terminal. What is the address the server received the message from? Which device/interface does this IP address belong to?

At this point, you have learned how to create a simple web app, leveraging HTML, performing rendering with Electron, using JavaScript to achieve some more advanced dynamic functionality, and finally having the Electron frontend communicate with the server. These steps have gotten you acclimated with some of the frontend development tools and technologies which will serve you well in the next steps of this lab. Feel free to play around more if you like, and when you feel ready, please proceed to the next step!

## Step 3: Develop Your Frontend

In the old days if you wanted to create a web-based frontend for your application you'd write HTML code. Over the years things became more dynamic, with backends dynamically generating HTML via perl and PHP, and the frontend getting more dynamic by actually supporting dynamic rendering code in the browser with Javascript. In some sense, Javascript kind of won the war<sup>5</sup> on how frontends are built and since frontends are so crucially important for so many applications it's good to have some experience in how to build them.

Now, it's atypical to write Javascript completely from scratch - there are a lot of libraries out there with code already written, and you should use these as they speed up your development a lot. Example libraries: D3.js, jQuery, anime.js, Some of these libraries are so extensive that they actually define the control flow of your application, these libraries are called "frameworks". Example frameworks: Vue.js, React.js, Angular.js, etc.

In this lab, we will be using ElectronJS to design your remote-control web app. ElectronJS is an open-sourced cross-platform desktop app development platform that starts gaining more and more attention nowadays. Lots of desktop apps you are using everyday (VS code, Atom, Slack, Discord, WhatsApp, Twitch, etc) are built using ElectronJS. The documentations are well written and tons of tutorials can be the guidance to your app development. For more information about ElectronJS, check out their website here <https://www.electronjs.org>.

---

<sup>5</sup> Something you can do in the future after you finish this lab, and if you want to get experience in alternate frontend technologies, is use Unity to create an AR-enabled control framework for your car. Eg you can do something like this: <https://www.youtube.com/watch?v=LbtXTCeWNXQ> - but instead of having a completely virtual car, create a virtual car that sits atop your real car, so your car becomes big and lifelike (and emits fire!) when you view it through your AR glasses. Since Bluetooth is a standard and you're developing an open API here your car needn't change and you can use existing C# libraries for Bluetooth.

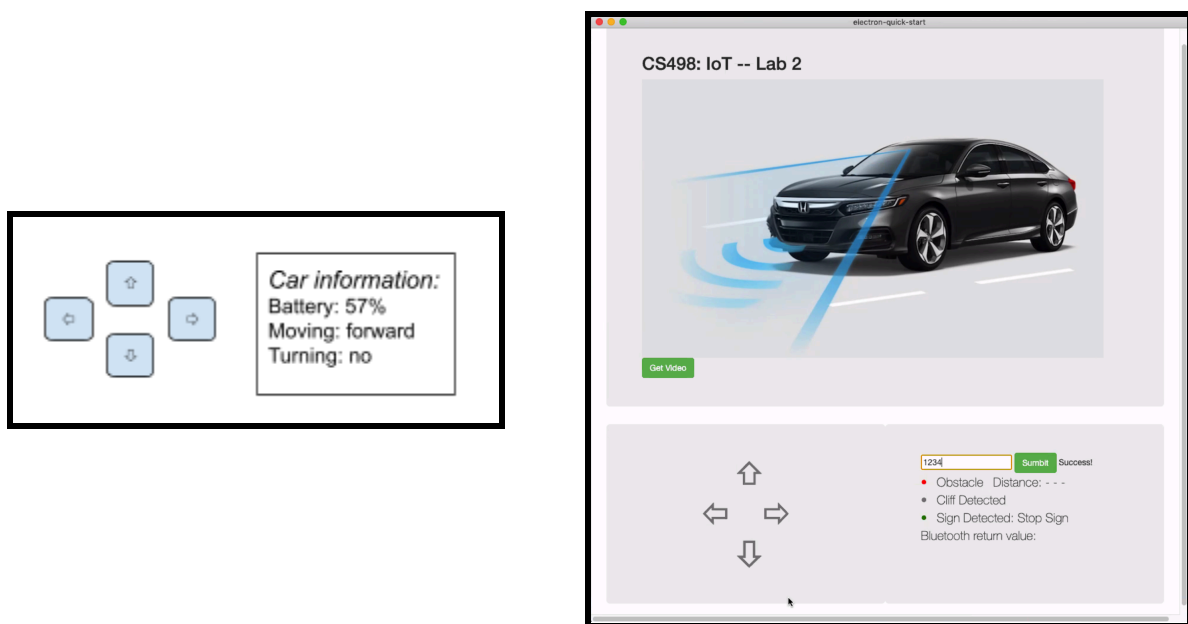
To install Electron on your desktop, please follow the instructions here<sup>6</sup>:

- <https://www.electronjs.org/docs/tutorial/development-environment>

After installing Electron, you can download the skeleton code we provided for you to help you get started:

<https://github.com/mccaesar/iot-labs/tree/master/iot-lab-2/electron>

Please take a close look at this code and try to understand it. You can extend it to build your own app. Your app should allow control of the car (to allow the car to go forward/backward and turn), as well as reading sensor data from the car and displaying it (e.g., battery life). Here is an example interface you might have for your UI:



**Figure 3: Example Smart Remote Control UIs (left: mockup view, right: screenshot of sample UI). You can have some buttons to remotely control your car, and a statistics pane to display information about the car's status. You don't have to follow exactly this. Feel free to go further and do additional things<sup>7</sup> if you like.**

To make sure you have Electron installed successfully, please check the following:  
(This

<sup>6</sup> If you like, you can use electron-reload, which is a nice library that can be integrated with electron for hot reload of UI, the ui changes would reflect as soon as we save the files.

<sup>7</sup> One thing you could do is purchase a Raspberry Pi Camera, mount it on your car, then stream video images back to Electron. Then you can drive the car remotely (e.g., from another room).

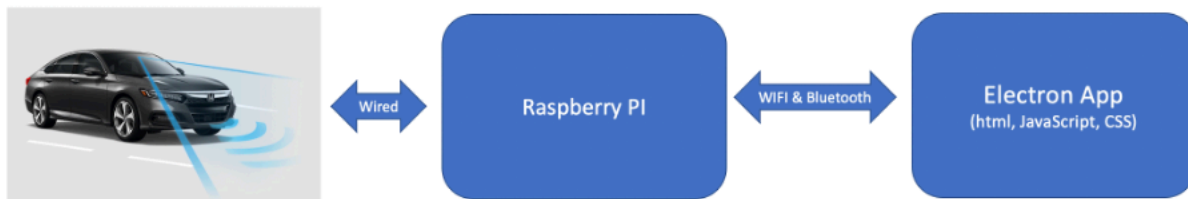
command should return the version of Node.js)

```
$ node -v
```

(This

command should return the version of npm)

```
$ npm -v
```



**Figure 4: Implementation architecture (component view).**

Within the provided

code directory, install Electron app packages. You only need to do this once.

```
$ npm install electron
```

To launch

the Electron app:

```
$ npm start
```

Start the

server on the Raspberry PI

```
$ python3 wifi_server.py
```

Modify the `index.js`, `index.html`, and expand `wifi_server.py` file. Think about what message you should send to the server in `index.js` and what message you should respond to the client in `wifi_server.py`. After receiving the message in `index.js`, update the corresponding values in `index.html`. You may change the code and frontend page based on your preference, but make sure you satisfy the minimum requirements.

# Submission and Grading

To submit your own work, you will create a **demo video** (under 10 mins) clearly showing the above steps working. Also, please give an overview of the code you wrote, show us your code and walk us through it a bit and how it works, so we can know your code is correctly written. Besides the demo video, you should also submit a **report** containing (briefly) your design thoughts and considerations for each part of the lab. Make sure that the lab report also includes contributions made by each of the group members in sufficient detail.

**Note:** Please include your GitLab repository link containing your complete codebase in your report.

The rubric for grading that you should follow is provided here:

1. **Is the background on wireless IoT understood?** Does the report provide a reasonable, correct, and thoughtful description of what protocols could/should be used for designing a real self-driving car platform? If so, give them 15 points for this item.
  - a. If what they write seems correct and reasonable, and expresses some knowledge taught in the Background section given in this lab - give them 15 points.
  - b. If what is written generally seems right but there is at least some clear technical incorrectness or lack of understanding - give them 8 points.
  - c. If nothing is written about this, or if what is written is largely incorrect - give them 0 points.
2. **Is the bluetooth connection working?** Was the bluetooth connection successfully established? Is data being exchanged over the connection? Can data be correctly sent and received over the connection? (The data exchange should be demonstrated by making the pi send car stats to your PC/Mobile App and having your PC/Mobile App send car control commands to the pi) If so, give them 5 more points.
  - a. If the bluetooth connection seems to be fully working, or at least mostly working aside from minor issues that don't substantially affect data transmission, give them 5-4 points.

- b. If the bluetooth connection is substantially broken, in a way that substantially prevents transmission, for example if transmission can only be done in one direction, or data is corrupted, but from looking at the code it looks like they largely followed the right steps, give them only 3 points here.
  - c. If the bluetooth connection is not working or has major problems, for example no data can be exchanged, but from looking at the code it looks like it should be working and largely the right steps were followed, give them only 2 points here.
  - d. If the bluetooth connection is not working at all, and the code seems largely wrong, but some reasonable attempt was made, give them only 1 points here.
  - e. If this step was not done, or there was no substantial attempt, for example if no or almost no code was written, give them 0 points for this item.
3. **Is the frontend working?** Is the web frontend working? Can it be used to control the car and display data? (WiFi should be used to exchange data bidirectionally) If so, give them 15 more points. (Please display at least 3 pieces of information, your choice)
- a. If the frontend appears to be running, if it displays some data from the car (any data that is sent from the car is ok), if it allows control of the car, then give them 10 points for this item.
  - b. If the frontend is lacking functionality, for example it allows control of the car but does not display sensor data, or vice versa, give only 7 points for this item.
  - c. If the frontend is lacking significant functionality, but runs at all in a web browser, and displays any sort of data or allows any sort of control of the car, and the code seems to be written generally in a correct way, give only 5 points for this item.
  - d. If this step was not done, or code was not described, or functionality of the web application was not demonstrated to any significant extent, give 0 points for this item.
4. Is there a video explaining the sections?
- a. 5 points for explaining code about communication with the pi car over wifi and demonstration.
  - b. 5 points for explanation of code for car moving and getting stats
  - c. 5 points for showing the car moving from web interface
  - d. 5 points for showing web interface getting stats from car

Submission/Points	Points
Step 1 - Background on wireless IoT	15
Step 2 - Bluetooth connection	0 (5 bonus points if working)
Step 3 - Frontend	15
Video	20
Peer review	0
<b>Total</b>	<b>50</b>

## Where to Go From Here

You now have the ability to add networking capabilities to your devices. While computing networking is a deep space and there is a lot to learn, what you can already do is quite powerful. Examples:

1. Make your apartment start playing a spotify playlist when you walk in the door (many cloud services, including Spotify, have REST APIs that you can call from any internet-connected device - they will often have some way to authenticate you, for example a web-based procedure for you to acquire a cryptographic token, which you can install on your device for this to work).
2. Build several more cars, mount nerf guns atop them, and have them surround your enemies before firing (look up "flocking" algorithms to coordinate the cars in a distributed and fault-tolerant manner over wireless and consider short-range protocols that leverage channel hopping to evade surveillance by your adversaries).
3. Perform distributed sensing - monitor table availability in a restaurant, monitor building infrastructure during earthquakes, monitor microclimate in your data center (consider using beacon-based algorithms like BLE to let your devices sleep as much as possible - have a leader plugged in that can collect data up from a control tree deployed over the nodes - you will learn more about this in the lectures).