Software Development Practice 1

Instructor: RSP <rawat.s@eng.kmutnb.ac.th>

Exploring Web Apps on Ubuntu / Raspberry Pi OS

Objectives

- Learn how to build and run a real-time web application using both Python (Quart) and Node.js (Express).
- Practice asynchronous programming using Python and Node.js.
- Understand the integration of WebSockets and MQTT for real-time data communication.
- Visualize system data (e.g., CPU temperature) using Plotly in a web browser.
- Compare backend implementation styles and async programming models in Python and Node.js.

Expected Learning Outcomes

After completing all tasks, students will be able to:

- Set up and run a simple real-time web application using either Python or Node.js.
- Use WebSockets and MQTT to transmit data between the server and client.
- Visualize data in the browser using Plotly.js.

Task 1: Running a Web App with Python or Nodejs as Backends

In this task, we will create and run a **Web app demo** on **Raspberry Pi OS** using two backend frameworks for comparison: **Nodejs (v20.x) with Express** and **Python (v3.11.x) with Quart**.

1. Create the following directory structure for the **Node.js project.**

```
./ws_plotly_node/
├── app.js
└── public/
    └── index.html
```

```
# Check the Node.js version (assume that Node.js is already installed).
$ node -v

# Create and enter the project directory.
$ mkdir -p ./ws_plotly_node && cd ./ws_plotly_node/
```

2. Install the necessary packages.

```
# Initialize the Node.js project.
$ npm init -y

# Install the following packages.
$ npm install express socket.io systeminformation
```

3. After adding / editing `app.js` and `public/index.html`, run the Web server and open the Web browser to see the page (use your RPi IP address, port 5000). To stop the server, press Ctrl+C.

```
$ node app.js
```

4. Create the following directory structure for the **Python project**.

```
./ws_plotly_quart_py/
  ├── app.py
  └── templates/
       └── index.html
```

5. Install the necessary Python packages (in a Python virtual environment).

```
$ pip3 install quart python-socketio[asgi] hypercorn
$ pip3 install plotly psutil aiomqtt
```

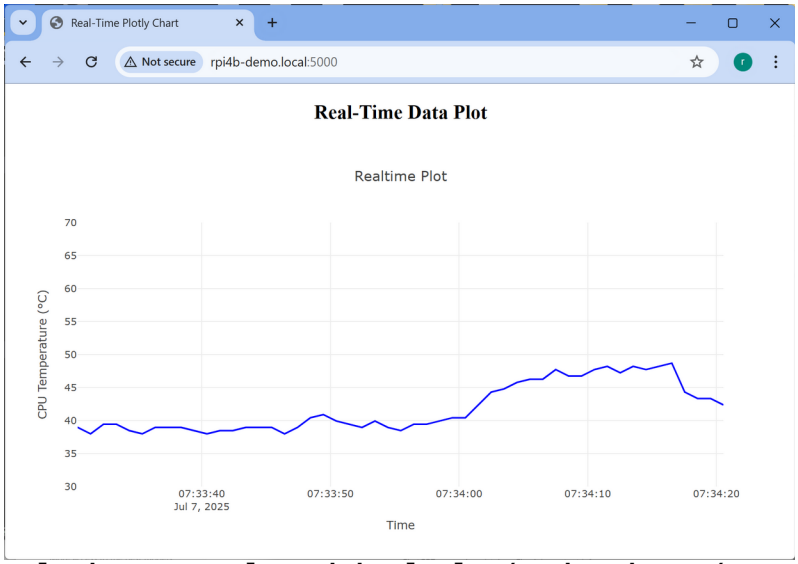
Note: The `asyncio` package is already included with Python 3.3 or higher.

6. After adding / editing `app.py` and `templates/index.html`, run the server. To stop the server, press Ctrl+C.

```
$ python ./ws_plotly_quart_py/app.py
```

7. Use the following commands to run **stress tests** (only for 30 seconds) on the RPi and observe changes in the CPU temperature.

```
$ sudo apt install stress-ng
$ stress-ng --cpu 4 --timeout 30s
```



Real-Time Data Plot with Plotlt / WebSockets / Quart

File: ws_plotly_node/app.js

```
const express = require('express');
const http = require('http');
const { Server } = require('socket.io');
const sysinfo = require('systeminformation');

// Create an Express app instance.
const app = express();
// Create an HTTP server for the Express app.
const server = http.createServer(app);
// Create a Socket.IO server attached to the HTTP server (port 5000).
const io = new Server(server);
const PORT = 5000;
// Serve static files from the 'public' folder.
app.use(express.static('public'));

app.get('/', (req, res) => {
    res.sendFile(__dirname + '/public/index.html');
});

io.on('connection', (socket) => {
    console.log(`Client ${socket.id} connected`);
    socket.on('disconnect', () => {
        console.log(`Client ${socket.id} disconnected`);
    });
});

async function send_cpu_temp() {
    let lastValue = 0.0;
    while (true) {
        try {
            const tempData = await sysinfo.cpuTemperature();
            let cpuTemp = tempData.main;
            if (cpuTemp === -1 || cpuTemp === null || isNaN(cpuTemp)) {
                cpuTemp = lastValue;
            } else {
                lastValue = cpuTemp;
            }
            const data = {
                x: Date.now() / 1000,
                y: cpuTemp
            };
            io.emit('data', data);
        } catch (err) {
            console.error('Error getting temperature:', err);
        }

        await new Promise(resolve => setTimeout(resolve, 1000));
    }
}

server.listen(PORT, () => {
    console.log(`Server listening on http://localhost:${PORT}`);
    send_cpu_temp();
});
```

File: ws_plotly_quart_py/app.py

```
import sys
import time
import asyncio
import socketio
from quart import Quart, render_template

# Create a Socket.IO server using
# the Asynchronous Server Gateway Interface (ASGI) framework.
sio = socketio.AsyncServer(async_mode='asgi')

# Create a Quart web application instance.
app = Quart(__name__)

# Combine Socket.IO and Quart into a single ASGI app.
sio_app = socketio.ASGIApp(sio, app)

# Used to keep active Socket.IO client session IDs (sio).
connected_clients = set()

# Define an asynchronous route handler serving an HTML page at the root URL.
@app.route('/')
async def index():
    return await render_template('index.html')

# Define an asynchronous function used to read CPU temperature and
# sends to WebSocket Socket.IO clients.
async def linux_send_cpu_temp():
    import psutil
    last_value = 0.0
    while True:
        temps = psutil.sensors_temperatures()
        cpu_temp = None
        name = 'cpu_thermal'
        if name in temps:
            entries = temps[name]
            if entries and hasattr(entries[0], 'current'):
                cpu_temp = float(entries[0].current)

        if cpu_temp is None:
            cpu_temp = last_value
        else:
            last_value = cpu_temp

        data = {'x': time.time(), 'y': cpu_temp}

        # Send the CPU temperature to all connected WebSocket clients.
        await sio.emit('data', data)
        await asyncio.sleep(1)
```

```
# Define the Socket.IO event handler for connection.
@sio.event
async def connect(sid, environ):
    if sid not in connected_clients:
        print(f'Client {sid} connected for the FIRST time')
        connected_clients.add(sid)
    else:
        print(f'Client {sid} reconnected')

# Define the Socket.IO event handler for disconnection.
@sio.event
async def disconnect(sid):
    print(f'Client {sid} disconnected')
    connected_clients.discard(sid)

if __name__ == '__main__':
    import hypercorn.asyncio
    from hypercorn.config import Config

    config = Config()
    config.bind = ["0.0.0.0:5000"]
    config.workers = 1

    if sys.platform.startswith("linux"):
        print("Linux")
        send_cpu_temp = linux_send_cpu_temp
    else:
        print("Other OSes are not supported yet!")
        sys.exit(-1)

    # Get The current asyncio event loop.
    loop = asyncio.get_event_loop()
    asyncio.set_event_loop(loop)

    # Run the send_cpu_temp() function running in the event loop,
    # as a background taks.
    loop.create_task(send_cpu_temp())

    # Start the ASGI server (Hypercorn).
    loop.run_until_complete(hypercorn.asyncio.serve(sio_app, config))
```

Note:

- You can also use the following Linux command in a separate console to terminate the Python app server.

```
$ kill -9 `pgrep -f app.py`
```

File: index.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Real-Time Plotly Chart</title>
  <script src="https://cdn.socket.io/4.8.0/socket.io.min.js"></script>
  <script src="https://cdn.plot.ly/plotly-latest.min.js"></script>
</head>
<body>
  <h2 style="text-align: center;">Real-Time Data Plot</h2>
  <div id="chart" style="width:100%;height:500px;"></div>

  <script>
    const socket = io();

    const data = [{
      x: [],
      y: [],
      mode: 'lines',
      line: {color: 'blue'}
    }];

    const layout = {
      title: 'Realtime Plot',
      xaxis: {title: 'Time'},
      yaxis: { title: 'CPU Temperature (°C)', range: [30, 70] }
    };

    Plotly.newPlot('chart', data, layout);

    socket.on('data', function(msg) {
      Plotly.extendTraces('chart', {
        x: [[new Date(msg.x * 1000)]],
        y: [[msg.y]]
      }, [0]);

      const maxPoints = 100;
      if (data[0].x.length > maxPoints) {
        Plotly.relayout('chart', {
           'xaxis.range': [
              data[0].x[data[0].x.length - maxPoints],
              data[0].x[data[0].x.length - 1]
            ]
        });
      }

      data[0].x.push(new Date(msg.x * 1000));
      data[0].y.push(msg.y);
    });
  </script>
</body>
</html>
```

Task 2: MQTT Communications

In this task, we will explore how to utilize the **MQTT protocol** to publish and subscribe messages using a **public MQTT broker**.

1. Create the following directory structure for the **Node.js project**.

```
./ws_mqtt_plotly_node/
├── app.js
└── public/
    └── index.html  <--- (use the same file as in Task 1)
```

2. Install the necessary packages for Node.js.

```
# Check the Node.js version.
$ node -v
```

Create and enter the project directory.

```
$ mkdir -p ./ws_mqtt_plotly_node && cd ./ws_mqtt_plotly_node/
# Initialize the Node.js project.
$ npm init -y
# Install necessary packages.
$ npm install express socket.io mqtt
```

3. After adding / editing `app.js` and `public/index.html`, run the server and open the Web browser to see the page.

```
$ node app.js
```

4. Install the Mosquitto MQTT client.

```
$ sudo apt install -y mosquitto-clients
```

5. Use the provided **Bash script** to publish messages to the MQTT broker. Run the command in a separate Linux terminal.

```
$ sudo apt install bc
$ chmod +x ./mqtt_pub_test.sh
$ ./mqtt_pub_test.sh
```

6. Create a project directory with the necessary files for the **Python project**.

```
./ws_mqtt_plotly_py/
  ├── app.py
  └── templates/
       └── index.html  <--- (use the same file as in Task 1)
```

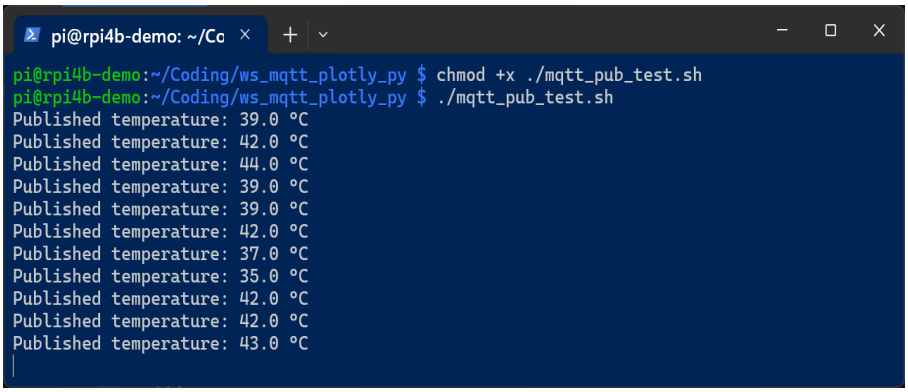
7. Test the Python project.

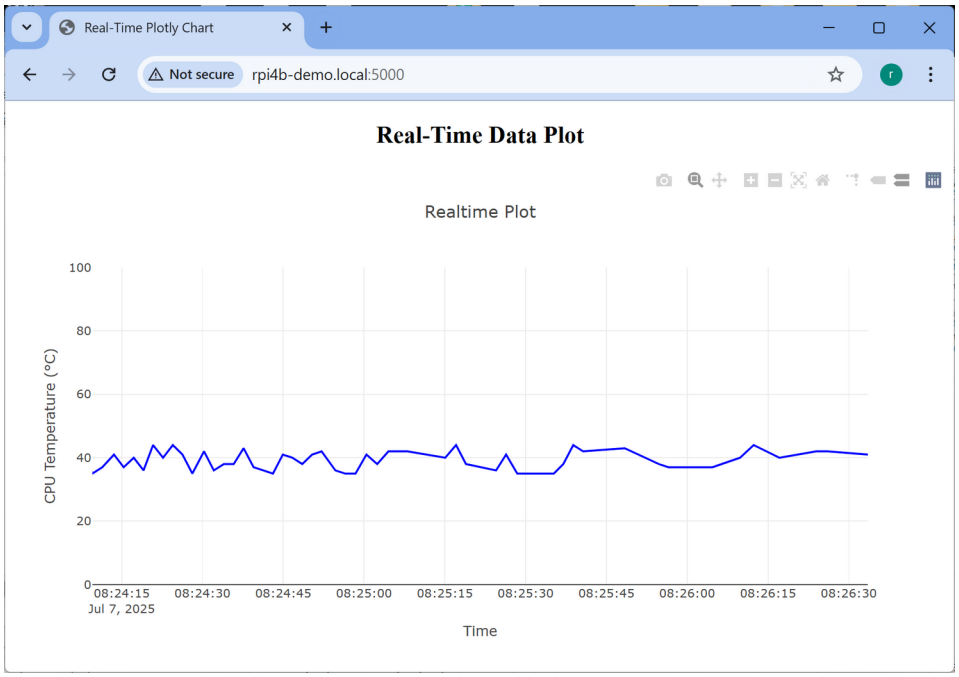
Note:

- Use a **unique topic** for publishing and subscribing messages.

File: mqtt_pub_test.sh

```
#!/bin/bash

BROKER="broker.hivemq.com"
TOPIC="rpi4b-demo/test/cpu/temp"
QOS=1

while true; do
  # Generate random float between 35.0 and 45.0
  VAL=$(echo "35 + ($RANDOM % 1000) / 100" | bc)
  TEMP=$(printf "%.1f" "$VAL")
  # Publish to MQTT broker
  mosquitto_pub -q $QOS -h $BROKER -t $TOPIC -m "$TEMP"
  # Log to console
  echo "Published temperature: $TEMP °C"
  # Wait 1 second
  sleep 1
done
```





File: ws_mqtt_plotly_node/app.js

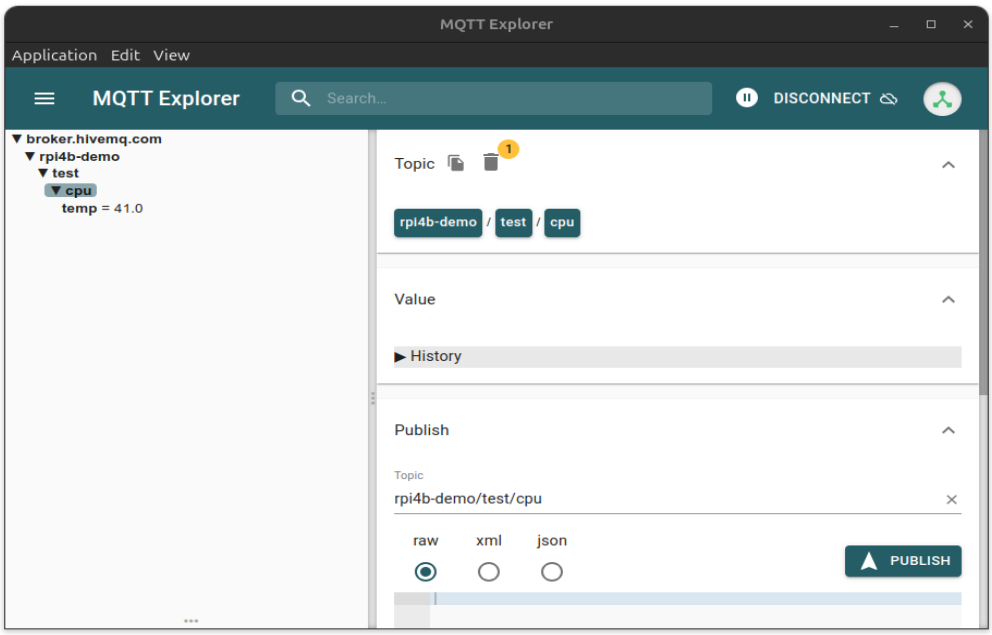
```
const express = require('express');
const http = require('http');
const { Server } = require('socket.io');
const mqtt = require('mqtt');
const path = require('path');

const MQTT_BROKER = 'mqtt://broker.hivemq.com';
const MQTT_TOPIC = 'rpi4b-demo/test/cpu/temp';

const app = express();
const server = http.createServer(app);
const io = new Server(server);
const PORT = 5000;

app.use(express.static(path.join(__dirname, 'public')));
// MQTT client setup and subscribe
const mqttClient = mqtt.connect(MQTT_BROKER);

mqttClient.on('connect', () => {
  console.log('Connected to MQTT broker');
  mqttClient.subscribe(MQTT_TOPIC, (err) => {
    if (err) {
      console.error('Failed to subscribe:', err);
    } else {
      console.log(`Subscribed to topic: ${MQTT_TOPIC}`);
    }
  });
});

mqttClient.on('message', (topic, message) => {
  if (topic === MQTT_TOPIC) {
    let payload = message.toString();
    let value = parseFloat(payload);
    if (!isNaN(value)) {
      const data = { x: Date.now() / 1000, y: value, };
      io.emit('data', data);
    } else {
      console.warn('Invalid payload received:', payload);
    }
  }
});

// Socket.IO connection handlers
io.on('connection', (socket) => {
  console.log(`Client ${socket.id} connected`);
  socket.on('disconnect', () => {
    console.log(`Client ${socket.id} disconnected`);
  });
});

server.listen(PORT, () => {
  console.log(`Server listening on http://localhost:${PORT}`);
});
```

File: ws_mqtt_plotly_py/app.py

```
import asyncio
import time
import socketio
from quart import Quart, render_template
from aiomqtt import Client, MqttError

MQTT_BROKER = "broker.hivemq.com"
MQTT_TOPIC = "rpi4b-demo/test/cpu/temp"

# Set up async Socket.IO and Quart
sio = socketio.AsyncServer(async_mode='asgi')
app = Quart(__name__)
sio_app = socketio.ASGIApp(sio, app)

connected_clients = set()

@app.route('/')
async def index():
    return await render_template('index.html')

# Subscribe to the MQTT broker.
async def mqtt_subscriber():
    try:
        async with Client(MQTT_BROKER) as client:
            await client.subscribe(MQTT_TOPIC)
            async for msg in client.messages:
                try:
                    payload = float(msg.payload.decode())
                    data = {"x": time.time(), "y": payload}
                    await sio.emit('data', data)
                except ValueError:
                    print("Invalid payload:", msg.payload)
    except MqttError as e:
        print("MQTT Error:", e)

@sio.event
async def connect(sid, environ):
    print(f'Client {sid} connected')
    connected_clients.add(sid)

@sio.event
async def disconnect(sid):
    print(f'Client {sid} disconnected')
    connected_clients.discard(sid)

if __name__ == '__main__':
    import hypercorn.asyncio
    from hypercorn.config import Config
    config = Config()
    config.bind = ["0.0.0.0:5000"]
    config.workers = 1
    loop = asyncio.get_event_loop()
    asyncio.set_event_loop(loop)
    loop.create_task(mqtt_subscriber())
    loop.run_until_complete(hypercorn.asyncio.serve(sio_app, config))
```

Extra Assignments for Tasks 1 & 2

1. If possible, try using **other public MQTT brokers**, such as **EMQX** or **Mosquitto**.
2. Use **MQTT explorer** to view MQTT messages.
3. Select and use other types of system information representing dynamic data for real-time visualization instead of CPU temperature.
   - Choose multiple sources (at least two) of system information for data visualization.
   - Use JSON-formatted strings for MQTT publishing and subscribing.
   - Modify the Python and Node.js code and index.html accordingly to plot data from all selected sources on the web page.



MQTT Explorer (on Ubuntu Linux)

Task 3: Scanning Nearby WiFi Networks.

In this task, we will explore how to write **Python** and **Node.js** code to scan nearby Wi-Fi networks **using Raspberry Pi SBC** and also **Ubuntu.**

1. Create a **Python script** and use the provided example (`wifi_scan_py/main.py`). This script uses the `asyncio` package to run the `nmcli` command as a subprocess.

2. Run the Python script and observe the output.

3. Create a **Node.js** script and use the provided example (`wifi_scan_node/main.js`). This script uses the `child_process` module to run the `nmcli` command as a subprocess.

4. Run the Node.js script and observe the output.

5. Run the following command in a separate terminal to force nmcli to rescan nearby Wi-Fi network periodically.

```
$ while true; \
  do echo "[Wi-Fi Rescan at $(date)]"; \
  sudo nmcli device wifi rescan; sleep 5; done
```

5. Create the following directory structure for the **Node.js** project for Web App (`ws_wifi_scan_js`).

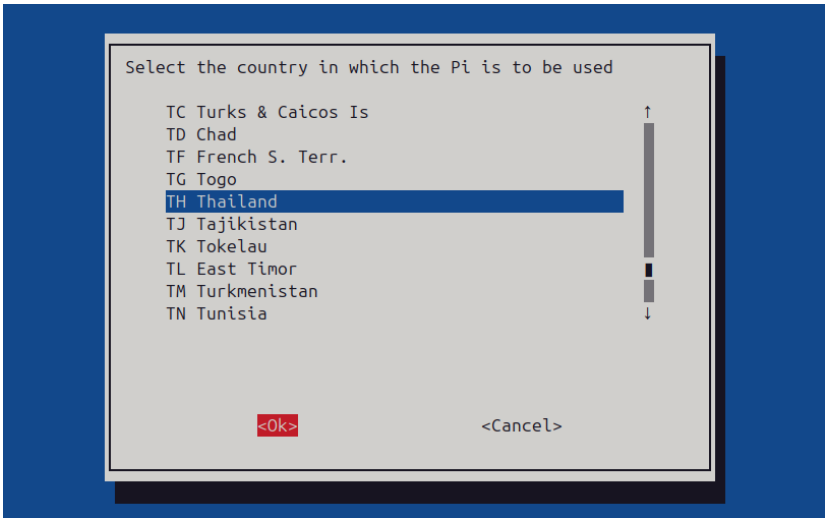
```
./ws_wifi_scan_js/
├── app.js
└── public/
    └── index.html
```

6. Run the Node.js server and observe the result displayed on the Web browser.

```
$ mkdir -p ./ws_wifi_scan_js/ && cd ./ws_wifi_scan_js/
$ npm init -y
$ npm install express socket.io
$ node app.js
```

Note:

- On the **Raspberry Pi**, make sure that your have enabled the Wi-Fi interface and set the country to TH (for Thailand) properly (use **raspi-config**).



File: wifi_scan_py/main.py

```
import asyncio
import json

async def scan_wifi():
    try:
        field_names = "SSID,BSSID,CHAN,SIGNAL,SECURITY"
        names = field_names.split(',')

        # Run nmcli asynchronously.
        proc = await asyncio.create_subprocess_exec(
            "nmcli", "-t", "-f", field_names, "device", "wifi", "list",
            stdout=asyncio.subprocess.PIPE,
            stderr=asyncio.subprocess.DEVNULL
        )

        stdout, _ = await proc.communicate()
        output = stdout.decode('utf-8')

        networks = []
        for line in output.strip().split("\n"):
            if not line.strip():
                continue
            line = line.replace(r"\:", "-")
            values = line.split(":")
            if len(values) == len(names):
                pairs = dict(zip(names, values))
                print(pairs)
                networks.append(pairs)

        return json.dumps(networks, indent=2)

    except Exception as e:
        return json.dumps({"error": str(e)})

async def main():
    try:
        while True:
            result = await scan_wifi()
            print(result)
            print("\n" + "=" * 40 + "\n")
            await asyncio.sleep(5)
    except KeyboardInterrupt:
        print("Terminated.")

if __name__ == "__main__":
    asyncio.run(main())
```

File: wifi_scan_node/main.js

```
const { exec } = require('child_process');
const util = require('util');

const execPromise = util.promisify(exec);
const FIELD_NAMES = "SSID,BSSID,CHAN,SIGNAL,SECURITY";
const FIELD_KEYS  = FIELD_NAMES.split(',');

async function scanWifi() {
    try {
        const networks = [];
        const { stdout } = await execPromise(
                           `nmcli -t -f ${FIELD_NAMES} device wifi list`);
        const lines = stdout.trim().split('\n');

        for (const line of lines) {
            if (!line.trim()) continue;
            const cleanLine = line.replace(/\\:/g, '-');
            const values = cleanLine.split(':');

            if (values.length === FIELD_KEYS.length) {
                const entry = Object.fromEntries(
                              FIELD_KEYS.map((key, i) => [key, values[i]]));
                console.log(entry);
                networks.push(entry);
            }
        }
        return JSON.stringify(networks, null, 2);

    } catch (err) {
        return JSON.stringify({ error: err.message });
    }
}

async function main() {
    try {
        while (true) {
            const result = await scanWifi();
            console.log(result);
            console.log('\n' + '='.repeat(40) + '\n');
            await new Promise(resolve => setTimeout(resolve, 5000));
        }
    } catch (err) {
        console.error('Terminated.', err);
    }
}

main();
```

File: ws_wifi_scan_node/app.js

```
const express = require('express');
const http = require('http');
const { Server } = require('socket.io');
const { exec } = require('child_process');
const path = require('path');

const app = express();
const server = http.createServer(app);
const io = new Server(server);

// Serve static contents.
app.use(express.static(path.join(__dirname, 'public')));

// Parse nmcli output
function parseNmcliOutput(output) {
  const fieldNames = ['SSID', 'BSSID', 'CHAN', 'SIGNAL', 'SECURITY'];
  const networks = [];

  output.trim().split('\n').forEach(line => {
    if (!line.trim()) return;
    line = line.replace(/\\:/g, '-');
    const values = line.split(':');
    if (values.length === fieldNames.length) {
      const entry = {};
      fieldNames.forEach((key, i) => (entry[key] = values[i]));
      networks.push(entry);
    }
  });

  return networks;
}

// Scan Wi-Fi networks.
function scanWifi() {
  return new Promise(resolve => {
    const fields = 'SSID,BSSID,CHAN,SIGNAL,SECURITY';
    const cmd = `nmcli -t -f ${fields} device wifi list`;
    exec(cmd, (error, stdout) => {
      if (error) {
        console.error('Wi-Fi scan error:', error.message);
        return resolve([]);
      }
      resolve(parseNmcliOutput(stdout));
    });
  });
}
```

```
// Handle Socket.IO clients
io.on('connection', socket => {
  console.log(`Client ${socket.id} connected.`);
  let active = true;

  socket.on('disconnect', () => {
    console.log(`Client ${socket.id} disconnected.`);
    active = false;
  });

  // Scan loop with 5-second interval
  async function scanLoop() {
    while (active) {
      const wifiList = await scanWifi();
      socket.emit('wifi_data', wifiList);
      await new Promise(resolve => setTimeout(resolve, 5000));
    }
  }

  scanLoop(); // Start scanning

});

// Start the server.
const PORT = 5000;
server.listen(PORT, () => {
  console.log(`Server running at http://localhost:${PORT}`);
});
```

File: ws_wifi_scan_node/public/index.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>WiFi Scanner</title>
  <script src="/socket.io/socket.io.js"></script>
  <style>
    table { border-collapse: collapse; width: 67%; margin: auto; }
    th, td { border: 1px solid #aaa; padding: 6px; text-align: center; }
    th { background-color: #eee; }
  </style>
</head>

<body>
  <h2 style="text-align: center;">Scanning Nearby WiFi Networks</h2>
  <table id="wifiTable">
    <thead>
      <tr>
        <th>SSID</th>
        <th>BSSID</th>
        <th>Channel</th>
        <th>Signal</th>
        <th>Security</th>
      </tr>
    </thead>
    <tbody></tbody>
  </table>

  <script>
    const socket = io();
    socket.on('wifi_data', function (data) {
      const tbody = document.querySelector('#wifiTable tbody');
      tbody.innerHTML = '';
      data.forEach(row => {
        const tr = document.createElement('tr');
        tr.innerHTML = `
          <td>${row.SSID}</td>
          <td>${row.BSSID}</td>
          <td>${row.CHAN}</td>
          <td>${row.SIGNAL}</td>
          <td>${row.SECURITY}</td>
        `;
        tbody.appendChild(tr);
      });
    });
  </script>
</body>
</html>
```

Extra Assignments for Task 3

1. Test the Node.js script on other machines with Ubuntu OS in the same location as the Raspberry Pi SBC, and compare the results.
2. Revise the Node.js script to limit the output to a limit of up to 10 Wi-Fi networks. Show only Wi-Fi networks with non-blank SSIDs (Wi-Fi names).
3. Apply CCS to colorize the table rows. For example, use alternating background colors to table rows for better readability.

Last update: 2025-07-07