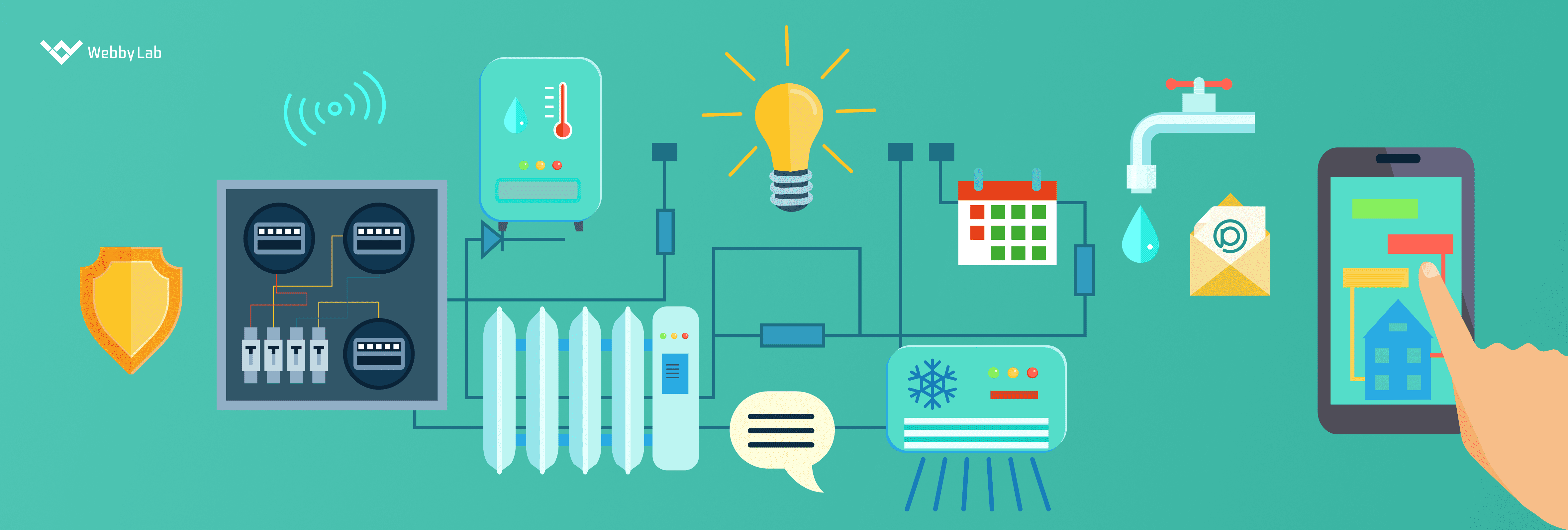
DOMAIN : INTERNET OF THINGS

TEAM NO : 09

PROJECT TITLE : SMART WATER MANAGEMENT

PHASE :03 SUBMISSION



To build a smart water management system . We have to collect and preprocess data effectively. By sing temperature sensors, ultrasonic sensors, water flow sensors, and a water flow monitor,this is how we load and preprocess the dataset:

**Sensor Data Collection:**

- Connect and configure your sensors (temperature, ultrasonic, water flow, etc.) to the ESP32 pins.

- Use appropriate libraries to collect data from these sensors.

**COLLECTING DATA FROM SENSORS**

**Temperature Sensors**: Collect temperature data to monitor ambient conditions. The code to collect the data from temperature sensor is given below

**CODE :**

**import os**

**import glob**

**import time**

**# Define the file path for reading temperature sensor data**

**sensor\_file = "/sys/bus/w1/devices/28-xxxxxx/w1\_slave" # Replace with your device's ID**

**def read\_temperature():**

**try:**

**with open(sensor\_file, "r") as sensor\_file:**

**lines = sensor\_file.readlines()**

**return parse\_temperature(lines)**

**except Exception as e:**

**print(f"Error reading temperature: {str(e)}")**

**return None**

**def parse\_temperature(lines):**

**if "YES" in lines[0]:**

**temperature\_line = lines[1].find("t=")**

**if temperature\_line != -1:**

**temperature\_string = lines[1][temperature\_line + 2:]**

**temperature\_celsius = float(temperature\_string) / 1000.0**

**return temperature\_celsius**

**return None**

**try:**

**while True:**

**temperature = read\_temperature()**

**if temperature is not None:**

**print(f"Temperature: {temperature}°C")**

**# Your main program logic can go here**

**time.sleep(2) # Delay to control the data collection frequency**

**except KeyboardInterrupt:**

**pass**

**Ultrasonic Sensors:** These sensors can be used to measure water levels in tanks or reservoirs. We’re also going to collect the data from ultrasonic sonic sensors as same as the temperature sensor.

**CODE :**

**import RPi.GPIO as GPIO**

**import time**

**# Define GPIO pins for the ultrasonic sensor**

**TRIG\_PIN = 18 # Trigger pin**

**ECHO\_PIN = 24 # Echo pin**

**# Set GPIO mode to BCM**

**GPIO.setmode(GPIO.BCM)**

**# Set up the GPIO pins**

**GPIO.setup(TRIG\_PIN, GPIO.OUT)**

**GPIO.setup(ECHO\_PIN, GPIO.IN)**

**def measure\_distance():**

**# Send a trigger pulse to the sensor**

**GPIO.output(TRIG\_PIN, True)**

**time.sleep(0.00001)**

**GPIO.output(TRIG\_PIN, False)**

**# Measure the time it takes for the echo to return**

**while GPIO.input(ECHO\_PIN) == 0:**

**pulse\_start = time.time()**

**while GPIO.input(ECHO\_PIN) == 1:**

**pulse\_end = time.time()**

**# Calculate the distance based on the time difference**

**pulse\_duration = pulse\_end - pulse\_start**

**distance = pulse\_duration \* 17150 # Speed of sound in cm/s**

**distance = round(distance, 2)**

**return distance**

**try:**

**while True:**

**# Measure water level**

**distance = measure\_distance()**

**print(f"Water level: {distance} cm")**

**# Your main program logic can go here**

**time.sleep(2) # Delay to control the data collection frequency**

**except KeyboardInterrupt:**

**pass**

**finally:**

**GPIO.cleanup()**

In this code:

1. We import the `RPi.GPIO` library and define the GPIO pins for the Ultrasonic Sensor (TRIG\_PIN and ECHO\_PIN).

2. We set up the GPIO pins and define a function `measure\_distance` to calculate the water level based on the time it takes for the ultrasonic pulse to return.

3. Inside the `while True` loop, you can measure and print the water level. You can integrate this data into your main program logic for further processing, analysis, or transmission to a central server.

**Water Flow Sensors and Monitor**: These sensors can track the flow rate of water.

**CODE :**

**import RPi.GPIO as GPIO**

**import time**

**# Define the GPIO pin to which the water flow sensor is connected**

**FLOW\_SENSOR\_PIN = 17**

**# Initialize GPIO**

**GPIO.setmode(GPIO.BCM)**

**GPIO.setup(FLOW\_SENSOR\_PIN, GPIO.IN, pull\_up\_down=GPIO.PUD\_UP)**

**# Initialize variables**

**total\_flow = 0**

**last\_time = time.time()**

**# Define a function to calculate flow rate**

**def calculate\_flow\_rate(channel):**

**global total\_flow, last\_time**

**total\_flow += 1**

**current\_time = time.time()**

**elapsed\_time = current\_time - last\_time**

**flow\_rate = total\_flow / elapsed\_time**

**last\_time = current\_time**

**print(f"Flow Rate: {flow\_rate} L/s")**

**# Add an event listener for the GPIO pin**

**GPIO.add\_event\_detect(FLOW\_SENSOR\_PIN, GPIO.FALLING, callback=calculate\_flow\_rate)**

**try:**

**while True:**

**# Your main program logic can go here**

**pass**

**except KeyboardInterrupt:**

**pass**

**finally:**

**GPIO.cleanup()**

In this code:

1. We import the `RPi.GPIO` library and set up the GPIO pin connected to the water flow sensor.

2. We initialize variables to keep track of the total water flow and the last recorded time.

3. We define a function, `calculate\_flow\_rate`, which is called every time the flow sensor detects a pulse. It calculates the flow rate and prints it.

4. We use `GPIO.add\_event\_detect` to add an event listener to the GPIO pin. When the flow sensor detects a falling edge (a pulse), the `calculate\_flow\_rate` function is called.

5. Inside the `while True` loop, you can add your main program logic to process the flow data or send it to a central server or cloud platform for monitoring and analysis.

Data preprocessing involves cleaning, transforming, and preparing the data for analysis and visualization. Here's how to preprocess the collected data:

**Combine Data:** Combine data from different sensors into a single dataset based on the timestamp. Ensure all timestamps align correctly.

**CODE :**

**import pandas as pd**

**# Sample data for each sensor**

**ultrasonic\_data = {**

**'timestamp': ['2023-01-01 00:00:00', '2023-01-01 01:00:00', '2023-01-01 02:00:00'],**

**'ultrasonic\_reading': [10.2, 11.1, 12.5]**

**}**

**temperature\_data = {**

**'timestamp': ['2023-01-01 00:00:00', '2023-01-01 01:00:00', '2023-01-01 02:00:00'],**

**'temperature': [22.5, 23.2, 24.1]**

**}**

**water\_flow\_data = {**

**'timestamp': ['2023-01-01 00:00:00', '2023-01-01 01:00:00', '2023-01-01 02:00:00'],**

**'flow\_rate': [5.3, 6.0, 5.8]**

**}**

**# Convert data to Pandas DataFrames**

**df\_ultrasonic = pd.DataFrame(ultrasonic\_data)**

**df\_temperature = pd.DataFrame(temperature\_data)**

**df\_water\_flow = pd.DataFrame(water\_flow\_data)**

**# Merge dataframes on the 'timestamp' column**

**merged\_data = pd.merge(df\_ultrasonic, df\_temperature, on='timestamp', how='inner')**

**merged\_data = pd.merge(merged\_data, df\_water\_flow, on='timestamp', how='inner')**

**# Display the combined dataset**

**print(merged\_data)**

In this code:

- We create sample data for ultrasonic, temperature, and water flow sensors, each with timestamps.

- We convert the data into Pandas DataFrames.

- We use the `pd.merge` function to merge the dataframes based on the 'timestamp' column, ensuring that all timestamps align correctly .

**Data normalization :** Data normalization is an essential step in data preprocessing for a smart water management system using IoT. It helps to ensure that data from different sensors or sources are on the same scale, which is crucial for accurate analysis and modelling

**CODE :**

**import pandas as pd**

**from sklearn.preprocessing import MinMaxScaler**

**# Sample dataset**

**data = {**

**'timestamp': ['2023-01-01 00:00:00', '2023-01-01 01:00:00', '2023-01-01 02:00:00'],**

**'temperature': [22.5, 23.2, 24.1],**

**'ultrasonic': [10.2, 11.1, 12.5],**

**'water\_flow': [5.3, 6.0, 5.8]**

**}**

**# Create a Pandas DataFrame**

**df = pd.DataFrame(data)**

**# Select columns to normalize (excluding the 'timestamp')**

**columns\_to\_normalize = ['temperature', 'ultrasonic', 'water\_flow']**

**# Initialize the MinMaxScaler**

**scaler = MinMaxScaler()**

**# Apply Min-Max scaling to the selected columns**

**df[columns\_to\_normalize] = scaler.fit\_transform(df[columns\_to\_normalize])**

**# Display the normalized DataFrame**

**print("Normalized Data:")**

**print(df)**

In this code:

1. We create a sample dataset with temperature, ultrasonic, and water flow data.

2. We create a Pandas DataFrame from the sample data.

3. We select the columns that need to be normalized, excluding the 'timestamp' column.

4. We initialize the `MinMaxScaler` from Scikit-Learning

5. We apply Min-Max scaling to the selected columns using `scaler.fit\_transform()`.

6. The normalized data is displayed as a Pandas DataFrame.

**Handling Missing Data:** Check for missing data and handle it appropriately, either by filling missing values or removing rows with missing data .

**CODE :**

**import pandas as pd**

**# Sample dataset with missing values**

**data = {**

**'timestamp': ['2023-01-01 00:00:00', '2023-01-01 01:00:00', '2023-01-01 02:00:00', '2023-01-01 03:00:00'],**

**'temperature': [22.5, None, 24.1, 23.0],**

**'ultrasonic': [10.2, 11.1, None, 12.5],**

**'water\_flow': [5.3, 6.0, 5.8, None]**

**}**

**# Create a Pandas DataFrame**

**df = pd.DataFrame(data)**

**# Check for missing data**

**missing\_data = df.isnull().sum()**

**print("Missing Data:")**

**print(missing\_data)**

**# Fill missing values with a specific value (e.g., 0)**

**df\_filled = df.fillna(0)**

**# Remove rows with missing data**

**df\_dropped = df.dropna()**

**# Display the DataFrames after handling missing data**

**print("\nDataFrame with Missing Values Filled:")**

**print(df\_filled)**

**print("\nDataFrame with Rows Dropped:")**

**print(df\_dropped)**

In this code:

1. We create a sample dataset with missing values (represented as `None`).

2. We use Pandas to create a DataFrame from the sample data.

3. We check for missing data using `df.isnull().sum()` and print the count of missing values for each column.

4. We fill missing values with a specific value (in this case, 0) using `df.fillna(0)`.

5. We remove rows with missing data using `df.dropna()`.

**Data Visualization**

You can visualize the preprocessed data to gain insights and monitor water management. Here's how you can create a simple line plot:

**CODE :**

**import matplotlib.pyplot as plt**

**import datetime**

**# Sample data - timestamps and water consumption values**

**timestamps = [datetime.datetime(2023, 1, 1, i, 0) for i in range(24)]**

**water\_consumption = [15, 20, 18, 22, 30, 35, 40, 42, 38, 32, 28, 25, 20, 18, 22, 30, 35, 40, 42, 38, 32, 28, 25, 20]**

**# Create a line plot**

**plt.figure(figsize=(12, 6))**

**plt.plot(timestamps, water\_consumption, marker='o', linestyle='-')**

**# Customize the plot**

**plt.title('Hourly Water Consumption')**

**plt.xlabel('Time')**

**plt.ylabel('Water Consumption (Liters)')**

**plt.grid(True)**

**# Format the x-axis for better readability**

**plt.gca().xaxis.set\_major\_formatter(plt.matplotlib.dates.DateFormatter('%H:%M'))**

**plt.gca().xaxis.set\_major\_locator(plt.matplotlib.dates.HourLocator(interval=1))**

**# Rotate x-axis labels for better visibility**

**plt.xticks(rotation=45)**

**# Show the plot**

**plt.tight\_layout()**

**plt.show()**

In this code:

- We import the Matplotlib library for data visualization.

- We define sample data with timestamps (hours) and corresponding water consumption values.

- We create a line plot using `plt.plot()` with markers and a line connecting the data points.

- We customize the plot by adding a title, labels for the x and y-axes, and grid lines.

- We format the x-axis to display hours and rotate the labels for better visibility.

- Finally, we display the plot using `plt.show()`.