Plantique

Senior Project.

Table of Contents

Frontend Structure

```
frontend/
├── node_modules/
├── src/
│   ├── components/
│   ├── App.jsx
│   ├── index.css
│   ├── main.jsx
├── .dockerignore
├── .env
├── .prettierrc.json
├── Dockerfile
├── eslint.config.js
├── favicon.ico
├── index.html
├── package-lock.json
```

```
├── package.json
└── vite.config.js
```

Backend Structure

```
backend/
├── controllers/
├── models/
├── node_modules/
├── python_model/
├── routes/
├── services/
├── .dockerignore
├── .env
├── app.js
├── Dockerfile
├── package-lock.json
└── package.json
```

API

- **Frontend Domain**: `plantique.veeraprachx.dev`
- **Backend Base API (Production)**: `api-plantique.veeraprachx.dev`

Available API:

Receive from the Arduino. (Data)

- api-plantique.veeraprachx.dev/environments
- api-plantique.veeraprachx.dev/plant-images

Send to the Arduino. (Command)

- api-plantique.veeraprachx.dev/fans
- api-plantique.veeraprachx.dev/foggys
- api-plantique.veeraprachx.dev/valves

Link for Mocking Data

Examples for **GET**, **POST**, **UPDATE**, and **DELETE** operations for `/environments` can be found here.

Examples for **GET**, **POST**, **UPDATE**, and **DELETE** operations for `/fans` can be found here

Examples for **GET**, **POST**, **UPDATE**, and **DELETE** operations for `/valves` can be found here

Examples for **GET**, **POST**, **UPDATE**, and **DELETE** operations for `/foggys` can be found here

(Python-based, but it can be converted to any language)

API: `/environments`

- Environment information.
- Available for (GET, POST, PUT, DELETE).
- The data to be received from the Arduino.

Below are the details of each field:

Field	Type	Required	Default Value	Description
`airTemp`	Number	Yes	N/A	The air temperature in degrees Celsius.
`airPercentHumidity`	Number	Yes	N/A	The air humidity percentage.
`soilTemp`	Number	Yes	N/A	The soil temperature in degrees Celsius.
`soilPercentHumidity`	Number	Yes	N/A	The soil humidity percentage.
`timestamp`	Date	No	`Date.now`	The timestamp when the data was recorded (defaults to now).

Example Usage

Below is an example JSON object based on the schema:

```
{
  "airTemp": 25.5,
  "airPercentHumidity": 60,
  "soilTemp": 18.2,
  "soilPercentHumidity": 45,
  "timestamp": "2025-01-27T10:00:00Z"
}
```

API: `/plant-images`

- Image of the plant.
- Available for (GET, POST, PUT, DELETE).
- The data to be received from the Arduino.

Below are the details of each field:

Field	Type	Required	Default Value	Description

Field	Type	Required	Default Value	Description
`name`	String	Yes	N/A	The name of the plant or a descriptive name for the image.
`image`	String	Yes	N/A	The image data, encoded in Base64 format.
`timestamp`	Date	No	`Date.now`	The timestamp indicating when the image was added.

Example Usage

Below is an example JSON object based on the schema:

```
{
  "name": "Rose Plant",
  "image": "...",
  "timestamp": "2025-01-27T10:00:00Z"
}
```

API: `/fans`

- Duration of time that the fans will be turned on.
- Available for (GET, POST, PUT, DELETE).
- The data to be sent to the Arduino.

Below are the details of each field:

Field	Type	Required	Default Value	Description
`time`	Number	Yes	N/A	The duration of time that the fans will be turned on.
`timestamp`	Date	No	`Date.now`	The timestamp indicating when the command is sent.

Example Usage

Below is an example JSON object based on the schema:

```
{
  "time": "20",
  "timestamp": "2025-01-27T10:00:00Z"
}
```

API: `/foggys`

- Duration of time that the foggy will be turned on.
- Available for (GET, POST, PUT, DELETE).
- The data to be sent to the Arduino.

Below are the details of each field:

Field	Type	Required	Default Value	Description
`time`	Number	Yes	N/A	The duration of time that the foggy will be turned on.
`timestamp`	Date	No	`Date.now`	The timestamp indicating when the command is sent.

Example Usage

Below is an example JSON object based on the schema:

```
{
  "time": "20",
  "timestamp": "2025-01-27T10:00:00Z"
}
```

API: `/valves`

- Duration of time that the valve will be turned on.
- Available for (GET, POST, PUT, DELETE).
- The data to be sent to the Arduino.

Below are the details of each field:

Field	Type	Required	Default Value	Description
`time`	Number	Yes	N/A	The duration of time that the valve will be turned on.
`timestamp`	Date	No	`Date.now`	The timestamp indicating when the command is sent.

Example Usage

Below is an example JSON object based on the schema:

```
{
  "time": "20",
  "timestamp": "2025-01-27T10:00:00Z"
}
```


Frontend development

Backend base API

The base backend API URL is stored in the environment variable `import.meta.env.VITE_API_BASE_URL`.

- **Local**: Points to `http://localhost:5000` for development on your local machine.
- **Production**: Points to `https://api-plantique.veeraprachx.dev`.

You can find it in the `.env` file.

.env file

If you want to add more environment variables, note the following:

- All environment variables must **prefix with `VITE_`** and be in **uppercase**.
  - For example: `VITE_ABCD`, `VITE_XYZ`.
- These variables can be accessed in `.jsx` code using `import.meta.env.[variable_name]`.
  - For example: `import.meta.env.VITE_ABCD`, `import.meta.env.VITE_XYZ`.

Example CRUD

Example for `GET` request with `/environment` api

```
import axios from "axios";

const API_URL = `${import.meta.env.VITE_API_BASE_URL}/environment`;

const response = await axios.get(API_URL);

const data = response.data;
```

Example for `POST` request with `/environment` api

```
import axios from "axios";

const API_URL = `${import.meta.env.VITE_API_BASE_URL}/environments`;

const newEnvironmentData = {
```

```
  airTemp: formData.airTemp, // number
  airPercentHumidity: formData.airPercentHumidity, // number
  soilTemp: formData.soilTemp, // number
  soilPercentHumidity: formData.soilPercentHumidity, //number
};
await axios.post(API_URL, newEnvironmentData);
```

Example for `PUT` request with `/environment` api

```
import axios from "axios";

const API_URL = `${import.meta.env.VITE_API_BASE_URL}/environments`;

const editId = formData.id; // Id of the item

const updatedEnvironmentData = {
  airTemp: formData.airTemp, // number
  airPercentHumidity: formData.airPercentHumidity, // number
  soilTemp: formData.soilTemp, // number
  soilPercentHumidity: formData.soilPercentHumidity, // number
};

await axios.put(`${API_URL}/${editId}`, updatedEnvironmentData);
```

Example for `DELETE` request with `/environments` api

```
import axios from "axios";

const API_URL = `${import.meta.env.VITE_API_BASE_URL}/environments`;

const editId = formData.id; // Id of the item

await axios.delete(`${API_URL}/${id}`);
```

IOT Development

MQTT basics

- MQTT is a lightweight messaging protocol commonly used in IoT projects.
- It works on a **publish/subscribe** model:
- **Publish**: A device sends data to a specific 'topic' on a broker.
    - In the code (and the project), the Arduino publishes sensor data to the topic `plantique/cGxhbnRpcXVl` on a public broker `broker.hivemq.com`.

- **Subscribe**: Other devices or backend services can subscribe to that topic on that broker.
    - When they subscribe, they will receive any messages that are published to it.

How to send data to backend

- In this project, the Arduino sends sensor data to a backend system using the MQTT protocol.
- The backend subscribes to a specific topic to receive the data.

Here's what you need to know:

1. MQTT Topic

Topic: `plantique/cGxhbnRpcXVl` This topic is used to publish sensor data.

2. Broker

The broker used is a public broker, `broker.hivemq.com`, which is free forever. (But others can look at the published data if they know the topic name we used.)

3. Data Format

The data must be sent in a **specific JSON** format so that the backend can correctly parse and process it.

The required structure is as follows:

- **context**: A string that categorizes the data (e.g., `environment`).
- **contextData**: An object containing the sensor readings. In this case, it is environmental data.

```
{
  "context": "environment",
  "contextData": {
      "airTemp": <number>,
      "airPercentHumidity": <number>,
      "soilTemp": <number>,
      "soilPercentHumidity": <number>
  }
}
```

Note:

- Currently, the only supported context for the backend is the `environment`.
- Other contexts (e.g., plant-image) will be implemented in the future once there is clarification on the required data.

Below is an example code snippet that constructs the JSON message with the `environment` context and publishes it to the MQTT topic `plantique/cGxhbnRpcXVl`:

```
// Define the MQTT topic that the backend subscribes to.
String topic = "plantique/cGxhbnRpcXVl";
```

```
// Simulated sensor values (in a real scenario, replace these with actual
sensor readings)
float airTemp = 25.0;               // Example air temperature in Celsius
float airPercentHumidity = 55.5;    // Example air humidity percentage
float soilTemp = 22.0;              // Example soil temperature in Celsius
float soilPercentHumidity = 60.0;   // Example soil humidity percentage

// Construct the JSON message following the required data structure
String message = "{"
                 "\"context\": \"environment\","
                 "\"contextData\": {"
                 "\"airTemp\": " + String(airTemp, 1) + ","
                 "\"airPercentHumidity\": " + String(airPercentHumidity, 1)
+ ","
                 "\"soilTemp\": " + String(soilTemp, 1) + ","
                 "\"soilPercentHumidity\": " + String(soilPercentHumidity,
1) +
                 "}}";

// Publish the JSON message to the MQTT broker on the specified topic
client.publish(topic.c_str(), message.c_str());
```

Explanation of the Example Code

1. **Topic Definition**:

- The variable topic is set to `plantique/cGxhbnRpcXVl`, which is the topic that the backend subscribes to in order to receive sensor data.

2. **Simulated Sensor Data**:

- The code uses hardcoded values for demonstration:

3. **JSON Message Construction:**

- The `message` variable is built as a JSON string that adheres to the required format:
  - The `context` field is set to `environment`.
  - The `contextData` object contains the sensor values.

4. **Publishing the Message**:

- The client.publish() function sends the JSON message to the MQTT broker under the defined topic.
- Since the backend is subscribed to this topic, it will receive the published data immediately.

Full code

- Below is the full code for sending `environment` data.
- Each section of the code has comments explaining its purpose.
- These comments will help you understand the flow and functionality of the code.

- You can run it directly on the Arduino IDE. (Don't forget to install the required libraries.)
- The only section you need to change is the WiFi credentials.
- Once you run it, mock data will be sent to the backend and displayed on the frontend.
- The logs can be checked in the serial monitor at 115200 baud.

```
#include <WiFiS3.h>          // Library for WiFi functions
#include <PubSubClient.h>    // Library for MQTT client operations

// WiFi credentials
const char* ssid = "";       // WiFi SSID (network name)
const char* password = "";     // WiFi Password

// MQTT Broker settings
const char* mqtt_server = "broker.hivemq.com";  // Public MQTT broker
address

// Define a WiFi client and an MQTT client
WiFiClient wifiClient;                // Creates a WiFi client
PubSubClient client(wifiClient);      // Creates an MQTT client using the
WiFi client

void setup() {
    Serial.begin(115200);             // Start serial communication at
115200 baud for debugging

    // Connect to WiFi network
    Serial.print("Connecting to WiFi");
    WiFi.begin(ssid, password);       // Start connecting to the WiFi
network
    while (WiFi.status() != WL_CONNECTED) {  // Loop until the connection
is established
        Serial.print(".");
        delay(1000);                  // Wait 1 second before checking
again
    }
    Serial.println("\nConnected to WiFi!");  // Confirm connection

    // Set up MQTT broker connection
    client.setServer(mqtt_server, 1883);  // Define the MQTT broker and
port (1883)

    // Connect to the MQTT Broker
    while (!client.connected()) {      // Keep trying until connected
        Serial.print("Connecting to MQTT...");
        if (client.connect("ArduinoClient")) {  // Try connecting with
client ID "ArduinoClient"
            Serial.println("✅ Connected to MQTT broker!");
        } else {
            Serial.print("❌ Failed, retrying...");
            delay(2000);              // Wait 2 seconds before retrying
```

```
        }
    }
}

void loop() {
    // Reconnect to the MQTT broker if the connection is lost
    if (!client.connected()) {
        client.connect("ArduinoClient");
    }

    // Generate random sensor values (for testing purposes)
    float airTemp = random(180, 300) / 10.0;           // Random air
temperature
    float airPercentHumidity = random(400, 800) / 10.0;  // Random air
humidity
    float soilTemp = random(150, 250) / 10.0;            // Random soil
temperature
    float soilPercentHumidity = random(300, 700) / 10.0; // Random soil
humidity

    // Define the topic to publish on
    String topic = "plantique/cGxhbnRpcXVl";

    // Construct the JSON message with the required structure:
    // {
    //   "context": "environment",
    //   "contextData": {
    //       "airTemp": <value>,
    //       "airPercentHumidity": <value>,
    //       "soilTemp": <value>,
    //       "soilPercentHumidity": <value>
    //   }
    // }
    String message = "{"
                     "\"context\": \"environment\","
                     "\"contextData\": {"
                     "\"airTemp\": " + String(airTemp, 1) + ","
                     "\"airPercentHumidity\": " +
String(airPercentHumidity, 1) + ","
                     "\"soilTemp\": " + String(soilTemp, 1) + ","
                     "\"soilPercentHumidity\": " +
String(soilPercentHumidity, 1) +
                     "}}";

    // Publish the message to the MQTT broker on the specified topic
    client.publish(topic.c_str(), message.c_str());
    Serial.println("📡 Published: " + message + " to " + topic);

    delay(2000);  // Wait 2 seconds before publishing the next message
}
```

Important Commands

1. Start the Development Environment

```
cd /plantique
docker-compose up --build
```

2. Stop the Development Environment

```
cd /plantique
docker-compose down
```

3. Resolve Container Activation Errors

If you encounter an error while activating the container, it is often due to conflicts with existing containers or images.

Use the following command to remove all existing containers and images:

```
cd /plantique
docker-compose down --rmi all
```

Then activate the container again

```
docker-compose up --build
```

4. Resolve Code Changes Not Showing in the Application

If you've updated your code but don't see the changes reflected in the application, it might be due to a caching issue with the Docker image or container.

Use the following command to rebuild the container:

```
cd /plantique
docker-compose down --rmi all
docker-compose up --build
```