**Lab-1 GIT**

**Basics:**

Q) What is version control? Why is it important?

Ans) Version control is a system that enables developers and teams to track and manage changes to their code, documents, and other digital assets over time. It provides a structured way to organize, document, and collaborate on projects by keeping a history of all changes made to files and directories. Version control systems (VCS) help developers work together efficiently and ensure the integrity and stability of their codebase.

**Version control is important because:**

**History and Tracking**: Version control systems maintain a complete history of every change made to a project. This means you can see who made a change, when it was made, and what exactly was changed.

**Backup and Recovery**: Version control acts as a reliable backup system. If files are accidentally deleted or corrupted, you can easily restore previous versions.

**Conflict Resolution**: When multiple developers modify the same code simultaneously, version control systems provide tools to help resolve conflicts.

Q) What's the difference between Git and GitHub?

Ans) **Git is a Version Control System (VCS):** Git is a distributed version control system that allows developers to track and manage changes to their source code or other text-based files.

Git is the version control system itself, which you install and use on your local machine.

**GitHub is a Hosting Service for Git Repositories:** GitHub is a web-based platform that provides hosting for Git repositories.

GitHub is a popular choice for hosting, sharing, and collaborating on Git repositories, especially in the open-source community and among development teams.

Q) Describe the Git workflow (add, commit, push, pull).

Ans) The Git workflow involves a series of commands and actions that developers use to track changes to their code, collaborate with others, and maintain a version-controlled codebase.

**Add (git add):**

Before Git records changes in your project, you need to specify which files or changes you want to include in the next commit. This is done using the git add command.

**Commit (git commit):**

The git commit command saves the staged changes to the Git repository and moves the project's history forward.

**Push (git push):**

The git push command is used to upload your local commits to a remote repository. This makes your changes accessible to others who have access to the same remote repository.

**Pull (git pull):**

The git pull command is used to fetch changes from a remote repository and merge them into your local branch automatically. It combines the git fetch (to retrieve changes) and git merge (to integrate changes) steps into a single command.

Q) What is a repository in the context of Git?

Ans) In the context of Git, a repository (also known as "repo") refers to a data structure that stores all the files, history, and metadata for a specific project. It is a fundamental concept in version control and is used to manage and track changes to a set of files and directories over time.

**Commits:**

Q) What is a commit in Git?

Ans) a commit is a concept that represents a snapshot of the entire codebase at a specific point in time. It's a record of the changes made to the files and directories in a Git repository. Each commit contains the following information:

1.Author Information.

2.Timestamp.

3.Commit Message.

4.Changes.

Q) How is each commit uniquely identified?

Ans) Each commit in Git is uniquely identified by a hash, which is a long string of characters. This hash is generated based on the contents of the commit and its parent commit(s), ensuring that no two commits will have the same identifier.

**Remote Repositories:**

Q) What is a remote repository in the context of Git?

Ans) In the context of Git, a remote repository (often simply referred to as "remote") is a repository that exists on a different location or server from your local Git repository. A remote repository in Git serves as a central hub for sharing, collaborating, and synchronizing code changes among multiple developers or teams.

Q) What are the default names that Git uses for the repository you cloned from and your local repository?

Ans) If we clone a repository, the command automatically adds that remote repository under the name “origin”. So, git fetch origin fetches any new work that has been pushed to that server since you cloned it.

Q)How do you synchronize changes from a remote repository to your local one, and vice versa?

Ans) **To Synchronize Changes from Remote to Local (Fetch and Merge)**

**Step 1: Fetch Remote Changes (**git fetch**)**

**Step 2: Merge Remote Changes into Your Local Branch (**git merge **or** git pull**)**

**To Synchronize Changes from Local to Remote (Push)**

**Step 1: Commit Your Local Changes (**git commit**)**

**Step 2: Push Local Commits to the Remote Repository (**git push**)**:

**GitHub Specifics:**

Q) What is a pull request?

Ans) A pull request is a feature in Git-based version control systems that allows developers to propose changes they've made in their own fork or branch of a repository to be merged into the original repository. It serves as a formal way to request code review, collaboration, and integration of changes into the main codebase. Pull requests are commonly used in collaborative software development to maintain code quality and facilitate team coordination.

Q)How do you 'fork' a repository on GitHub, and why might you want to?

Ans) To 'fork' a repository on GitHub, you can click the "Fork" button on the top right of the repository's page. Forking creates a personal copy of the repository under your GitHub account. You might want to fork a repository to:

**Contribute:** Make changes to someone else's project.

**Experiment:** Test changes or explore new ideas without affecting the original.

**Collaborate:** Collaborate on a project with others, using forks as separate workspaces.

**Create a backup:** Safeguard a snapshot of a project for future reference.

Q) How can you use GitHub to collaborate on open-source projects?

Ans) To collaborate on open-source projects using GitHub:

**Fork the Repository:** Fork the project's repository to create your own copy.

**Clone Your Fork:** Clone your forked repository to your local machine.

**Create a Branch:** Create a new branch for your work.

**Make Changes:** Make and commit your changes locally.

**Push Changes:** Push your branch with changes to your fork on GitHub.

**Create a Pull Request:** Open a pull request to propose your changes to the original repository.

**Discuss and Review:** Collaborate with project maintainers and contributors through

**Collaboration and Best Practices:**

Q) Why is it important to write clear commit messages?

Ans) Clear commit messages are important because they:

Enhance Readability.

Aid Debugging.

Document History.

Improve Maintainability.

Q) When collaborating with others, why might it be important to frequently pull the latest changes?

Ans) Frequently pulling the latest changes is important when collaborating because it helps:

**Stay Synchronized:** Ensure your local copy is up to date with others' contributions.

**Prevent Conflicts**: Reduce the likelihood of code conflicts when multiple people are making changes.

**Facilitate Integration:** Simplify the process of merging your changes into the latest codebase.

**Enhance Collaboration:** Promote real-time collaboration and coordination among team members.

**LAB-2**

# Installing Mosquitto, running Mosquitto pub-sub clients, and publishing and receiving messages

Download and Install Mosquitto:

Download the Mosquitto MQTT broker from the official website.

https://mosquitto.org/download/

Install Mosquitto by following the installation wizard instructions.

Locate Mosquitto Installation Directory:

By default, Mosquitto may be installed in C:\Program Files\mosquitto.

To verify that Mosquitto has been added to your system path, open a new command prompt window (old command prompt windows won’t reflect the change) and type mosquitto -v. This should print the version of Mosquitto to the console if Mosquitto was added to the system path successfully.

By following these steps, you should be able to invoke Mosquitto from any location within the command line on your Windows system.



# MQTT broker in MQTT publish and subscribe model

The MQTT broker is a hub for communication between clients. It obtains messages from publishers and distributes them to the clients based on the topics they are subscribed to. This allows us to keep the workload balanced and maintain the linear growth of the connection count if the number of clients increases.

# MQTT Quality of Service for the messages

**QoS0** – the default level of quality of service that does not guarantee the delivery of the message. The publisher sends the message to the MQTT broker, not more than once, and the broker does not acknowledge the receipt of it. You can use this level when you can accept the message loss.

**QoS1** – the quality of service level that guarantees the delivery of the message. The publisher sends the message more than once until the MQTT broker acknowledges the receipt of it. You can use this level when the message must be delivered and you don’t have a problem with the subscriber receiving the message more than once.

**QoS2** – the quality of service level that guarantees the processing of the message only once. The publisher sends the message and stores it until a double handshake between the sender and receiver has accomplished a s.c. double handshake to acknowledge receipt of it. You can use this level when the message must be processed without any duplicates.

**LAB-4**

# Blinking LED:-

// Declare a variable for the LED pin

int ledPin = 13;

// The setup function runs once when you press reset or power the board

void setup() {

// Initialize the digital pin as an output for the LED.

pinMode(ledPin, OUTPUT);

// Initialize serial communication with a baud rate of 9600.

Serial.begin(9600);

}

// The loop function runs over and over again forever

void loop() {

// Turn the LED on

digitalWrite(ledPin, HIGH);

// Wait for one second

delay(1000);

// Turn the LED off

digitalWrite(ledPin, LOW);

// Wait for one second

delay(1000);

// Send a message to the serial monitor on your computer

Serial.println("LED is blinking");

}

**LAB-5**

# Interfacing DHT sensor with Arduino –

// Read DHT11 sensor and send serially to PC

#include <DHT.h> // Include Adafruit DHT11 Sensors Library

#define DHTPIN 7 // DHT11 Output Pin connection

#define DHTTYPE DHT11 // DHT Type is DHT11

DHT dht(DHTPIN, DHTTYPE); // Initialize DHT sensor

void setup () {

dht.begin();

Serial.begin(9600); // To see data on serial monitor

}

void loop (){

float H = dht.readHumidity(); //Read Humidity

float T = dht.readTemperature(); // Read temperature as Celsius

// Check if any reads failed and if exit

if (isnan(H) || isnan(T)){

Serial.println("Failed to read from DHT sensor!");

return;

}

// Combine Humidity and Temperature into single string

String dhtData = String(H) + "," + String(T);

Serial.println(dhtData);

delay(2000); // Wait two seconds between measurements

}

# LAB-6

# Gathering DHT sensor data and visualizing it using an app built by Node-RED



