Lecture - 04

REST (Representational State Transfer) - Short Note

- REST is an **architectural style** based on **web standards** and the **HTTP protocol**.

- The term **"Representational State Transfer"** refers to transferring the **representation** (like JSON, XML, etc.) of a **resource's state** between client and server.
- In REST, **everything is treated as a resource**, which is identified by a **global ID** (usually a URL/URI).
- REST uses standard **HTTP methods** to operate on resources:
  - **GET** (retrieve / Select),
  - **POST** (create / Insert),
  - **PUT** (update),
  - **DELETE** (remove).
- RESTful services are **stateless,** meaning each request from a client contains all the information needed for processing.
- **Statelessness** ensures high **scalability** and **fault tolerance**, making it suitable for **distributed systems**.
- **REST server** provides access to resources; **REST client** uses this interface to interact.
- A resource can have multiple **representations**, such as **JSON, XML, HTML**, etc.

☑ What does **"GET method should not alter the state"** mean?

In REST and HTTP, **GET** is intended to be a **read-only** operation. You open a book to read a page — you **see the content**, but you **don't write or change anything**. That's what **GET** is for. **GET** requests are used to **retrieve data** (like fetching a user profile, list of products, blog post, etc.).

✕ Example of what NOT to do with GET: GET /deleteUser/123

☑ Correct usage of GET: GET /users/123

Use sub-resources for relation:

If a resource is related to another resource, use sub-resources

GET /cars/711/drivers/ (Returns a list of drivers for car 711)

GET /cars/711/drivers/4 (Returns driver #4 for car 711)

Use HTTP headers for serialization formats:

❑ Content-Type defines the request format.

❑ Accept defines a list of acceptable response formats.

Versioning is important:

❑ Use a simple ordinal number and avoid dot notation, such as 2.5

❑ We are using the URL for the API versioning starting with the letter “v” /blog/api/v1

Paging

GET /cars?offset=10&limit=5   What it means:

- `GET`: You're **requesting data** (not modifying anything).
- `/cars`: You're accessing the **cars resource**.
- `?offset=10&limit=5`:
  - `offset=10`: **Skip the first 10** items.
  - `limit=5`: **Return the next 5** items.

Sorting

GET /cars?sort=-manufacturer,+model.

This returns a list of cars sorted by descending manufacturer and ascending models.

Filtering:

Use a unique query parameter for all fields.

❑ GET /cars?color=red (Returns a list of red cars)

❑ GET /users?name=tom (Returns a list of users whose name matches tom.

CoAP

The Constrained Application Protocol (CoAP) is a specialized web transfer protocol for use with constrained devices (such as micro-controllers) and constrained networks in the Internet of Things.

machine-to-machine (M2M) applications - smart energy and building automation.

Architecture Of CoAP

❑ It is specified in RFC 7252.

❑ It is an open IETF standard.

❑ It is a very efficient RESTful protocol.

❑Easy to proxy to/from HTTP.

❑It is an embedded web transfer protocol

❑ It uses an asynchronous transaction model.

❑ UDP is binding with reliability and multicast support.

❑ GET, POST, PUT, and DELETE methods are used.

❑ Uses a subset of MIME types and HTTP response codes.

❑ Uses built-in discovery mechanism.

❑URI- Uniform Resource Identifier is supported

❑It uses a small and simple 4-byte header

❑Supports binding to UDP, SMS, and TCP

❑ DTLS-based (Datagram Transport Layer Security) PSK (Pre-Shared Key), RPK(Raw Public Key), and certificate security are used.

MQTT – Message Queuing Telemetry Transport

MQTT is a **lightweight, event-driven protocol** ideal for IoT.
It supports **one-to-many** communication using a **publish/subscribe** model via a central **broker**.

✅ MQTT is an **event-based IoT middleware protocol** designed for **lightweight communication**, especially useful in **low-bandwidth**, **low-power**, or **unstable networks**, like in **IoT systems**.

📌 Key Features:

- **Event-Based**:
  Communication is **triggered by events** (e.g., a sensor detects motion → sends data).
- **One-to-Many Communication**:
  It uses a **publish-subscribe model**, allowing **one device** (publisher) to send a message to **many subscribers**.

🔁 How it works:

1. **Publisher**: Sends messages on a **topic**.
2. **Broker**: Central server that **receives** and **routes** messages.
3. **Subscriber(s)**: Devices or apps that **listen** to specific topics.

🧠 Example:

- A temperature sensor publishes:

```
bash
CopyEdit
Topic: home/room1/temp
Message: 26°C
```

- Multiple devices (e.g., mobile app, display screen, logging service) subscribed to that topic will **all receive the message** — this is **one-to-many**.

📡 MQTT – Lightweight Event-based IoT Middleware

- ✅ **Lightweight protocol** – minimal overhead, great for constrained devices.
- ✅ **Event-based communication** – uses **publish/subscribe** model.
- ✅ **Supports one-to-many** communication via **topics**.
- ✅ Runs **over IP using TCP** – reliable delivery of messages.
- ✅ Ideal for **IoT and embedded systems** (e.g., **Arduino**, ESP32).
- ✅ Works well over:
  - **Low-bandwidth** networks.
  - **Intermittent or unstable** connections.
  - **Long-range** wireless communication.

🔄 Publish / Subscribe Model – Key Characteristics

1. **Decoupling of Senders and Receivers**
   - Publishers and subscribers **don't know about each other**.
   - Promotes **flexibility**, **scalability**, and **loose coupling**.
2. **One-to-Many Communication**
   - A **single published message** can be delivered to **multiple subscribers**.
   - Efficient for **broadcasting events** to many devices or services.
3. **Dynamic Subscription**
   - Subscribers can **join or leave** at any time.
   - The set of subscribers can **change based on message type** or **application state**.

📦 MQTT Message Delivery – 3 Levels of Quality of Service (QoS)

1. **QoS 0 – At most once**
   - The message is sent **only once**.
   - **No guarantee** it will arrive.
   - **Fastest**, but **messages can be lost**.
   - Like: "I yelled, maybe they heard me."
2. **QoS 1 – At least once**
   - The message is **guaranteed to arrive**, but it **may be received more than once**.
   - **Safer**, but can cause **duplicates**.
   - Like: "I'll keep yelling until I know someone has heard me — even if I have to yell twice."
3. **QoS 2 – Exactly once**
   - The message is guaranteed to arrive **just once** — **no loss, no duplicates**.
   - **Most reliable**, but **slower**.
   - Like: "I'll shake hands to make sure we both agree — once and only once."

⚖️ Simple Rule:

Higher QoS = More reliability, but **slower performance** lowers the performance

---

Lecture -03

⚠️ Challenges in Embedded Systems Programming

Unlike general-purpose programming (like building apps on a PC), **embedded systems** have some **unique difficulties**:

⏱️ 1. Real-Time vs Non-Real-Time Systems

- Some embedded systems must respond **within a strict time**.

Types of Real-Time:

- **Hard Real-Time**:
  Must respond **exactly on time** – missing the deadline can cause failure.
  Example: Airbag system in a car
- **Soft Real-Time**:
  Best to respond on time, but small delays are okay.
  Example: Video streaming
- **Firm Real-Time**:
  Missing deadlines is bad, but the system **won't crash**.
  Example: Data logging in sensors

🔋 2. Constrained Resources

- Limited **CPU power**, **battery life**, and **energy use**.
- Must be efficient to run on **tiny devices**.

💾 3. Limited Memory

- Small memory for:
  - 📥 **Data operations** (RAM)
  - 💿 **Program storage** (ROM/Flash)

🔀 4. Limited Multitasking

- May not support full **multithreading** or **multi-processes**.
- Often just runs **one task at a time** or uses **basic scheduling**.

🧠 Summary:

Embedded systems programming is **tougher** because you're working with **tight timing**, **low memory**, and **limited computing power** — everything must be efficient and reliable.

🔧 IoT Prototyping Boards – Two Main Types

IoT boards are often divided into **two families** based on their **hardware and software capabilities**:

1️⃣ Microcontroller-Based Boards (MCU Boards)

- 🧠 Simple, small, and low-power
- Runs one program at a time (no operating system)
- Great for **sensing**, **controlling**, and **real-time tasks**

✅ *Examples:*

- **Arduino Uno**
- **ESP8266 / ESP32**
- **STM32**

2️⃣ Single-Board Computers (SBCs)

- 💻 Mini computers with full **operating systems** (like Linux)
- Can **multitask**, run complex software, support **networking, GUIs**, etc.
- More powerful but uses more energy

✅ *Examples:*

- **Raspberry Pi**
- **BeagleBone**
- **NVIDIA Jetson Nano**

🧠 Summary:

- Use **microcontroller boards** for small, real-time control tasks.

- Use **SBCs** when you need more computing power, an OS, or multitasking.

🔌 Arduino Analog Input

🎚️ Resolution

- Resolution is the **number of steps** used to measure a voltage.
- Higher resolution = more **precise** readings.

Bit	States (Steps)
8-bit	256
10-bit	1024 (**Arduino's default**)
32-bit	4,294,967,296

- For Arduino (10-bit):
  - Voltage range = **0 to 5V**
  - Smallest detectable change = **5V / 1024 = ~4.8mV**
  - Maximum sampling speed = **10,000 samples/second**

⚡ Arduino PWM (Pulse Width Modulation)

💡 What is PWM?

- Arduino **digital pins** can only turn **fully ON (5V)** or **fully OFF (0V)**.
- **PWM** turns the pin ON and OFF very fast (many times per second).
- The device connected sees this as an **"average" voltage** (like 1.5V, 2.5V, etc.).

🎛️ Example:

- If the pin is ON **50% of the time**, it feels like it's getting **2.5V** (half of 5V).
- Useful for **controlling LED brightness**, **motor speed**, etc.

🧠 Summary:

- **Analog input** lets Arduino read real-world signals (like temperature or light).
- **PWM** lets Arduino simulate **analog output** using digital pins.

🧃 The Raspberry Pi Board

- 🖥️ It is a **Single-Board Computer (SBC)**
- Uses a **Broadcom System-on-Chip (SoC)**
- Inside the SoC, it has:
  - 🧠 An **ARM-compatible CPU** (Central Processing Unit)
  - 🎮 An **on-chip GPU** (Graphics Processing Unit)

💡 Key Features:

- Works like a **mini computer**
- Can run a **full operating system** (like Linux)
- Used in **IoT projects, learning programming, robotics, media centers**, and more

🧃 The Raspberry Pi Board – GPIO Features

🔌 **GPIO** stands for **General Purpose Input/Output** —
It lets the Raspberry Pi **connect with and control** other devices like sensors, LEDs, motors, etc.

📍 Key Features:

- 🧮 **40 Digital Pins**
  Used for input/output signals (high/low = 3.3V/0V)
- 🎛️ **PWM (Pulse Width Modulation)**
  - Controlled **by software**
  - Used for things like LED brightness and motor speed control
- 📡 **UART (Universal Asynchronous Receiver Transmitter)**
  - For **serial communication**
  - Used to connect to GPS modules, Bluetooth, etc.
- 🔄 **I2C (Inter-Integrated Circuit)**
  - For connecting **multiple devices** (sensors, displays) using only **2 wires**
- 🔁 **SPI (Serial Peripheral Interface)**
  - High-speed communication between the Pi and peripherals (like ADCs, displays)

🔧 Arduino – Advantages:

- ✅ **Robustness** – No operating system, fewer chances of software failure
- ✅ **Low Power Consumption** – Ideal for battery-powered or energy-efficient systems
- ✅ **Lower Cost** – Very affordable for simple hardware projects
- ✅ **Real-Time Performance** – Executes code immediately, good for precise timing tasks
- ✅ **Easy to Use** – Great for beginners in electronics and embedded systems

---

🧃 Raspberry Pi – Advantages:

- ✅ **Powerful & Multitasking** – Can run multiple programs at the same time
- ✅ **Built-in Networking** – Comes with Ethernet, Wi-Fi, and Bluetooth
- ✅ **Full OS Functionality** – Supports Linux and other operating systems
- ✅ **Supports Multiple Languages** – Python, C++, Java, etc.
- ✅ **USB, HDMI, Camera Support** – Can connect to screens, peripherals, and more

---

🧠 Summary: Raspberry Pi vs Arduino

- 🧃 **Raspberry Pi**: Best for **complex, software-heavy projects** with networking and multitasking
- 🔧 **Arduino**: Best for **simple, hardware-focused projects** that need reliability and low power

🔌 I2C Protocol (Inter-Integrated Circuit)

- ✅ **Used in Embedded Systems** to connect **one or more master devices** to **one or more slave devices**.
- 🔁 **Bidirectional Communication** – Data flows **both ways** between master and slave.
- 🔗 **Two-wire Serial Bus**:
  - **SCL (Serial Clock Line)** – Carries the clock signal
  - **SDA (Serial Data Line)** – Carries the actual data
- 🧠 **Master Controls Communication**:
  - Master sends a signal to start communication
  - Master writes data to or reads data from slave
  - Each **slave device has a unique address**
- 🕒 **Synchronous Protocol**:
  - Communication is **clock-based**, using the SCL line
- ⚡ **Data Transfer Speeds**:
  - **Standard Mode**: up to **100 kbps**
  - **Fast Mode**: up to **400 kbps**
  - **Fast Mode+**: up to **1 Mbps**
  - **Ultra-Fast Mode**: up to **5 Mbps**

🧠 Summary:

I2C is a simple and efficient way to connect multiple devices (like sensors, displays, etc.) using just **two wires**, making it ideal for **low-speed, short-distance communication** in embedded systems.

🔄 SPI Protocol (Serial Peripheral Interface)

- ✅ **Used in Embedded Systems** for **high-speed data exchange** between devices.
- 🧠 **Master–Slave Architecture**:
  - One **Master** controls the communication.
  - One or more **Slaves** respond when selected.

🔌 Key Signals in SPI (Total: 4 Wires):

1. **SCLK (Serial Clock)**
   – Generated by the **Master** for timing/synchronization.
2. **MOSI (Master Out, Slave In)**
   – Carries data **from Master to Slave**.
3. **MISO (Master In, Slave Out)**
   – Carries data **from Slave to Master**.
4. **SS / CS (Slave Select / Chip Select)**
   – **Activated by the Master** (pulled LOW) to choose a slave device.

📡 Features of SPI:

- 🔄 **Full-Duplex Communication** – Data can be sent and received at the same time.
- ⚡ **Very High-Speed** – Data rates can exceed **100 MHz**.
- 🧩 **Multiple Slaves** supported, each needs a **separate SS line**.

🧠 Summary:

SPI is a fast and simple protocol used for **high-speed** communication between microcontrollers and peripherals like **sensors, displays, memory chips**, etc.

Lecture-3

🌐 IoT Communication Considerations

☑ 1. Communication with the Outside World

- How an IoT device talks to external systems (e.g., servers, cloud) affects the **network architecture**.
- Example: Wi-Fi, Bluetooth, Zigbee, Cellular, etc.

☑ 2. Technology Choice Affects Hardware and Cost

- The selected communication method decides the **hardware needed** (e.g., Wi-Fi chip, Bluetooth module).
- It also affects the **power use, size, and price** of the device.

☑ 3. No One-Size-Fits-All Networking Solution

- Because IoT devices are used in **many different scenarios** (e.g., smart homes, agriculture, healthcare, factories),
  one networking system **can't meet all needs**.
- Each use case might require **a different network approach** (like low power, long range, or fast speed

🌐 Complexity of Networks – Key Factors

☑ 1. Growth of Networks

- As the number of connected devices increases, networks become more **complex to manage** and maintain.
- Scalability becomes a major challenge.

☑ 2. Interference Among Devices

- Many devices using **similar frequencies** (like Wi-Fi or Bluetooth) can **interfere with each other**, causing communication issues.

☑ 3. Network Management

- More devices mean more effort to **monitor, configure, secure, and maintain** the network.
- Requires smart management tools and protocols.

✅ **4. Heterogeneity in Networks**

- Devices may use **different communication technologies** (e.g., Zigbee, LoRa, Wi-Fi).
- Making them work together creates **compatibility and integration challenges**.

✅ **5. Protocol Standardization**

- Lack of **common standards** across devices and networks leads to **incompatibility**.
- Standardization is key for smooth communication and interoperability.

📶 RFID – Radio-Frequency Identification

- ✅ **RFID = Radio-Frequency Identification**
- ✅ It stores **digital data** in small devices called **RFID tags**.
- ✅ A special device called a **reader** reads the data.
- ✅ **Similar to barcodes**, but more advanced.
- ✅ Data from the tags is saved in a **database**.
- ✅ **Main advantage:**
  → **RFID tags don't need to be in direct line-of-sight** to be read
  (unlike barcodes or QR codes).

⚙️ Working Principle of RFID

✅ **RFID comes from AIDC**

- AIDC = *Automatic Identification and Data Capture*
- It helps to **identify objects**, **collect data**, and **send it to computers** — all with **little or no human help**.

✅ **AIDC usually uses wires,**
But…

✅ **RFID uses **radio waves** to do the same job — wirelessly!

✅ **Main parts of an RFID system:**

1. **RFID Tag or Smart Label** – stores the data
2. **RFID Reader** – reads the tag's data
3. **Antenna** – helps send and receive radio signals

🧠 **Memory Tip:**

RFID = Wireless AIDC with tags, readers, and antennas. No touching needed!