جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

أكاديمية كاوست
KAUST ACADEMY

Internet of Things (IoT)

DAY 4 Out to the internet: MQTT, the cloud, and chat control

August 2 to 6, 2026

KGSP Summer Enrichment Program 2026
KAUST Academy, KAUST

Salah Abdeljabar
salah.abdeljabar@kaust.edu.sa

🗺️ On Today's Agenda

1. **Telemetry**: what to send, how often, and what if the link drops
2. 🔧 **Project 9**: live temperature and humidity on a page that updates itself
3. **MQTT**: the broker, topics, and getting a message delivered
4. 🔧 **Project 11**: publish to a cloud dashboard, and be commanded back
5. **The reference architecture**: the map, now that you have built one
6. 🔧 **Project 12**: read the sensor and switch the light by chat, from anywhere
7. **Security**: who is allowed to connect, and what your token is worth
8. 🎓 **The capstone brief**: form pairs, sketch an idea

By the end your board is reachable from anywhere, not just from this room.

Telemetry 

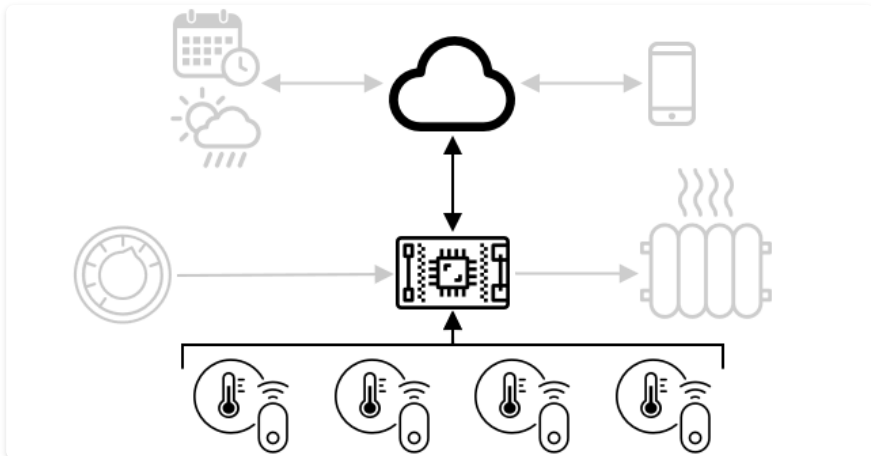
Sending sensor data from device to cloud

What is Telemetry?

From the Greek for *"measuring remotely"*: gathering sensor data and sending it to the cloud, usually as **JSON**.

```
{ "light": 300, "temperature": 22.5 }
```

The first telemetry device (1874) sent Mont Blanc weather to Paris over wires.



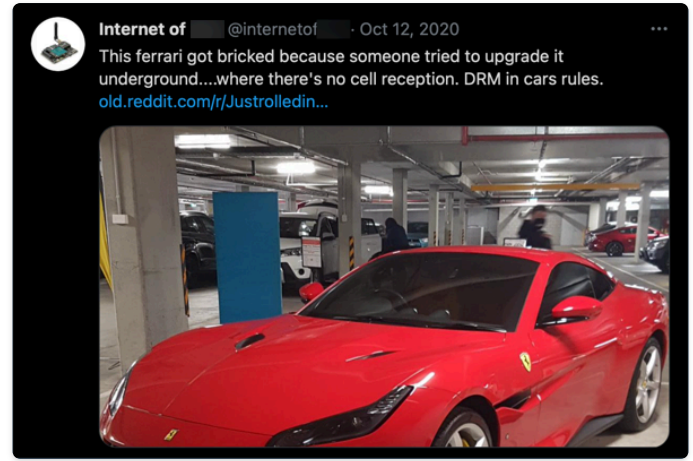
Frequency & Connectivity

How often to send?

Too often → wastes power, bandwidth, and money. **Too rarely** → you miss events. Match the rate to the job.

Caution

Connections drop. Smart devices should still work **offline**: a thermostat keeps deciding locally.



No signal, no cloud.

PROJECT 9 · SENSOR + WEB

DHT11 Temperature & Humidity Web Server

Read a DHT11 and show live temperature & humidity on a page that refreshes every 10 seconds.

Builds on: Project 8's polling pattern, now fetching sensor readings instead of a button state.

⏱️ ~25 min 📊 Intermediate 🔖 Prerequisite: Project 5

⬇ Download the sketch

Save `Project_9_ESP32_DHT11_Web_Server.ino`, then open it in the Arduino IDE (setup: **Foundation 3**). Needs the DHT, Adafruit Unified Sensor, ESP Async

Project 9 · DHT11 Web Server 🌡️

Live temperature and humidity on a web page that updates itself.

- Read a real environmental sensor through a library
- Handle a failed reading with `isnan()` instead of a nonsense number
- Fill page placeholders with live values via a template processor
- Compare a digital sensor's numbers with Project 2's raw ADC voltage

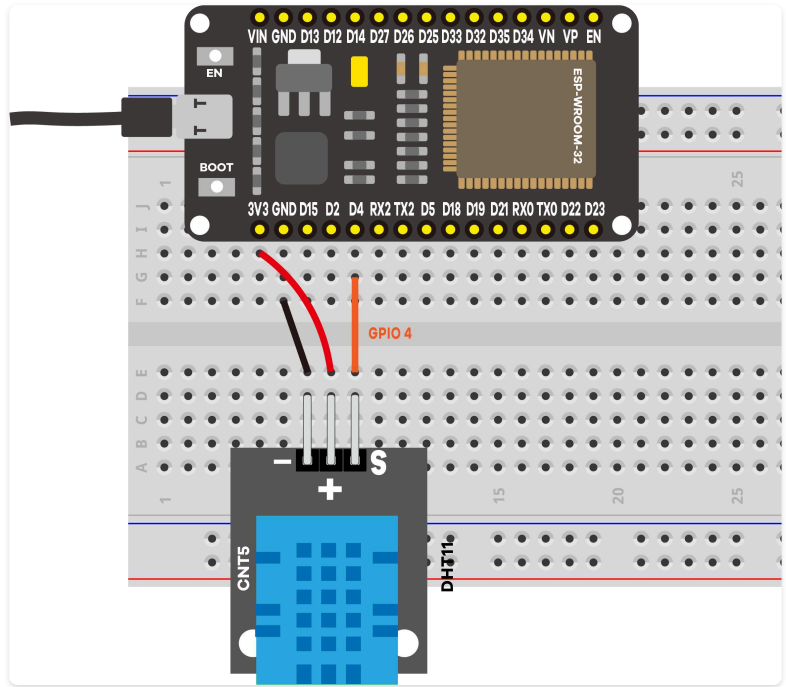
Key pin: DHT11 `4`

A Digital Sensor

The **DHT11** reports **temperature (°C)** and **relative humidity (%)** over a single data wire.

- **S** (signal) → **GPIO 4**, **+** → **3.3 V**, **–** → **GND**
- A 3-pin module with the pull-up built in, so no extra resistor

> **Note**
>
> Contrast Project 2: an **analog** sensor gives a voltage you must interpret; a **digital** sensor hands you finished numbers.



The Library Does the Hard Part

```
DHT dht(DHTPIN, DHTTYPE);          // pin + type + protocol code, bundled

float t = dht.readTemperature();   // degrees Celsius, already converted
if (isnan(t)) return "--";         // failed read: show a dash, not garbage
return String(t);                  // text, because it goes into a web page
```

- The DHT11 speaks a one-wire **pulse protocol**; `dht.readTemperature()` hides the microsecond timing you would otherwise write by hand.
- `isnan()` catches a failed reading. The library returns `NaN`, which is not equal to anything, so it needs its own test.
- A DHT11 fails now and then (a loose wire, or being polled too fast), so a fallback matters.

Never Blank, Never Stale

The first page load is filled by a **template processor**; after that the browser **polls** each value every 10 seconds.

```
String processor(const String& var){
  if (var == "TEMPERATURE") return readDHTTemperature(); // fills %TEMPERATURE%
  if (var == "HUMIDITY")    return readDHTHumidity();
  return String();
}
```

```
setInterval(function () {                    // every 10 s
  xhttp.open("GET", "/temperature", true);   // innerHTML swaps just the number
}, 10000);
```

- Placeholders make the first load carry real values; polling keeps them fresh.
- **10 s** matches the DHT11's roughly one-reading-per-second rate; polling harder gains nothing.

A Live Dashboard

The readings update on their own. Breathe near the sensor and the humidity climbs.

What you learned:

- A digital sensor hands you finished numbers; a **library** hides a fiddly protocol
- Always check for a failed read: a dash beats a wrong number
- Sensors have a **sampling rate**; match your refresh to it
- Project 11 sends the same readings to the **cloud**, to watch from outside the building

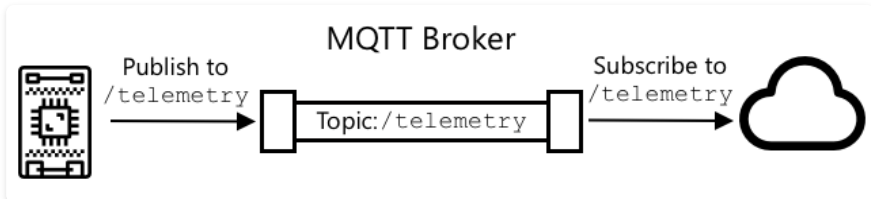
MQTT 

The standard messaging protocol for IoT

One Broker, Many Clients

MQTT was designed in **1999** to monitor oil pipelines, later open-sourced by IBM.

- All clients connect to a single **broker**
- Messages route by named **topics**, not to clients directly
- Topics are **hierarchical**: `/device-id/telemetry`
- **Wildcards**: subscribe to `/telemetry/*`



Reliability Features

Quality of Service (QoS)

- **0**: at most once (fire & forget)
- **1**: at least once (acknowledged)
- **2**: exactly once (two-step handshake)

Try it free

Public test broker: **test.mosquitto.org**
(no account needed, but not private!).

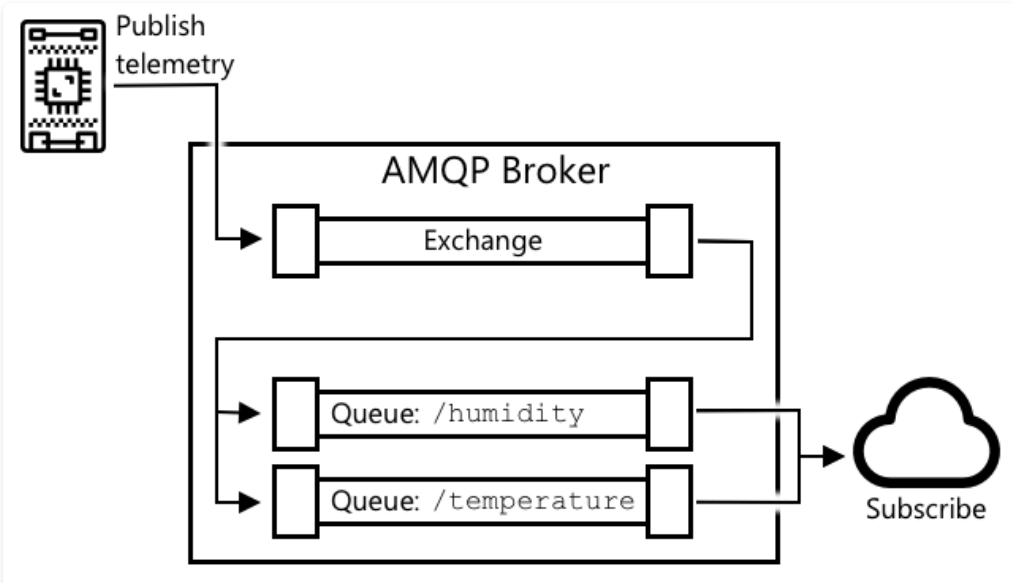
More guarantees

- **Retained**: broker keeps the last value
- **Keep-alive**: checks the connection is live
- **TLS** + username/password or certificates

A Closer Look at AMQP

AMQP splits the broker into an **exchange** and **queues**:

- The device publishes into the **exchange**
- The exchange **routes** each message into one or more queues
- Subscribers read from the queues, and a message waits until someone takes it
- Heavier than MQTT, but you get guaranteed delivery and richer routing



MQTT hands a message to whoever is listening now. AMQP holds it until it is collected.

PROJECT 11 · CLOUD / MQTT

MQTT + Ubidots: Cloud Dashboard and Control

Send DHT11 readings to a live cloud dashboard, and switch an LED or relay back from anywhere.

Builds on: Project 9's DHT11 reading and Project 7's actuator, now published to and controlled from the cloud instead of a local page.

⏱️ ~35 min 📊 Intermediate 🔖 Prerequisite: Project 9

⬇ Download the sketch

Save `Project_11_ESP32_MQTT_Ubidots.ino`, then open it in the Arduino IDE. Needs the **Ubidots ESP32 MQTT**,

Project 11 · MQTT + Ubidots ☁️

Publish sensor data to a cloud dashboard, and control an output from it.

- Contrast a web server's request/response model with MQTT's publish/subscribe
- Use the shared vocabulary: broker, topic, publish, subscribe
- Send real sensor readings to a cloud service and see them plotted
- Receive a command back through a callback function

Key pins: DHT11 `4`, output `26`

Why MQTT

Request / response (Projects 5 to 9)

- Your phone must be on the **same Wi-Fi**
- It has to keep **asking** the board for data
- Does not scale past the local network

Publish / subscribe (MQTT)

- Devices **publish** to a **broker**; others **subscribe**
- Lightweight, and works **from anywhere** with internet
- Nobody has to poll: the broker routes messages

This is the full IoT loop: DHT11 readings go **to** the cloud, and a dashboard switch comes **back** to the board.

Broker, Topic, Publish, Subscribe

Term	Meaning	Here
Broker	The server that routes every message	Ubidots hosts it
Publish	Send a value to a topic	ESP32 sends temperature and humidity
Subscribe	Ask to receive a topic's values	ESP32 listens for the `led` command
Topic	The address a message is filed under	One per device and variable

`DHT11` → `ESP32` publish → `Ubidots broker` → `Dashboard (you watch)`

`LED` ← `ESP32` ← subscribe `Ubidots broker` ← `Switch widget (you control)`

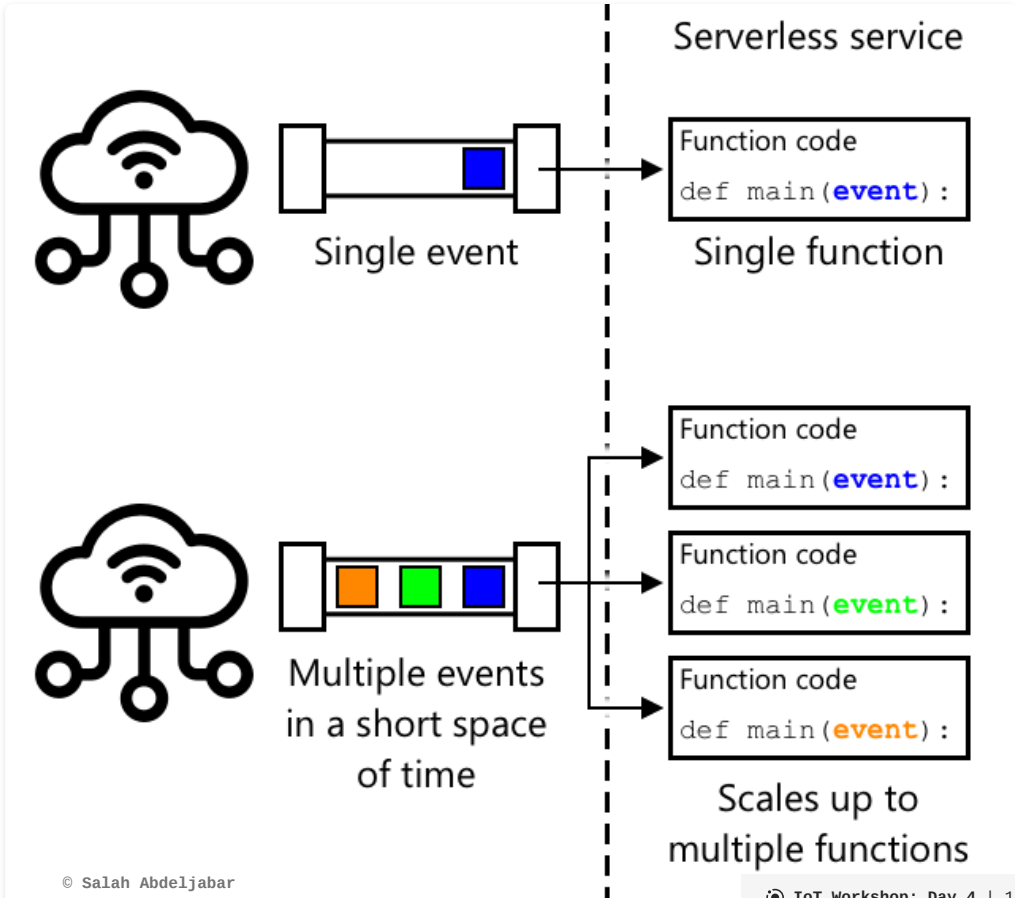
What the Cloud Does With It ☁️

Turning a stream of telemetry into insight

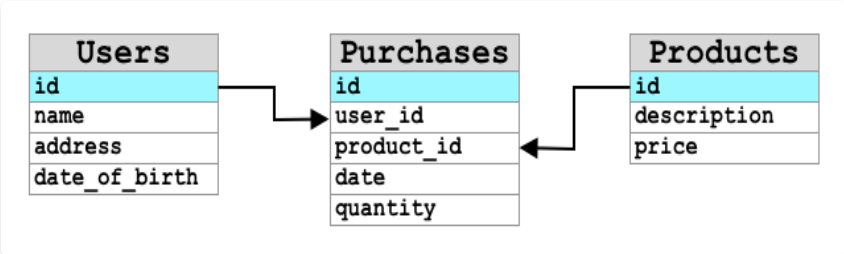
Serverless: Code Without a Server

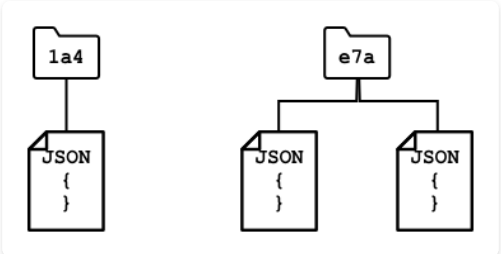
You write **one function** that handles **one message**. The platform runs it whenever a message arrives.

- No server to provision, patch, or keep running
- One message → one function run
- A burst of messages → many copies at once
- Pay only for the runs, so an idle device costs nothing



Where the Data Lands





SQL: fixed columns, rows linked between tables. Great when every reading has the same shape and you want to query across them.

NoSQL: documents keyed by id, each holding whatever JSON it likes. Great when devices send different fields.

Tip

IoT telemetry is often stored **raw** first and cleaned later: storage is cheap, and the question you'll want to ask next year hasn't been thought of yet.

The Cloud Talks Back: callback()

```
void callback(char *topic, byte *payload, unsigned int length) {
  float v = atof(value);              // raw payload -> number (copied safely first)
  digitalWrite(LED_PIN, v >= 0.5);    // dashboard switch -> the board's LED
}

ubidots.setCallback(callback);        // hand the function over
ubidots.subscribeLastValue(DEVICE_LABEL, VAR_LED);   // listen for `led`
```

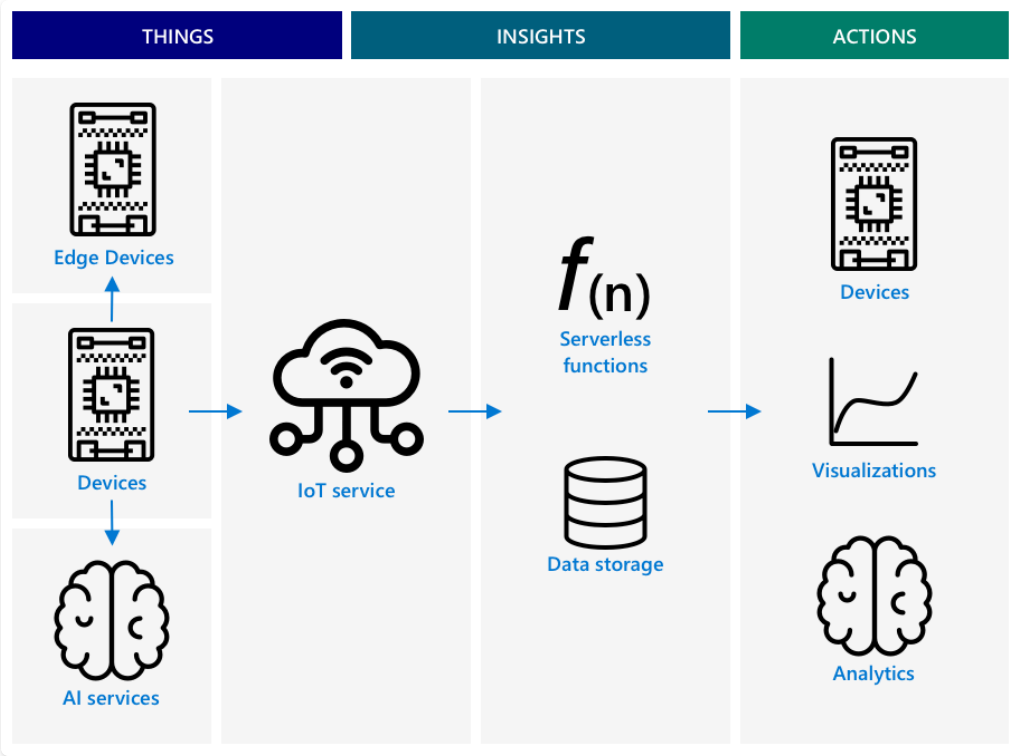
- You never call `callback()` yourself; the **library calls it** when a subscribed value arrives. That inversion is the callback pattern behind most event-driven networking.
- `v >= 0.5`, not `v == 1`: a switch may send `1`, `1.0`, or `1.000000`, and comparing floats for equality is a bad habit.
- `subscribeLastValue()` also pulls the **current** value, so a switch left on is picked up at boot.

A loop() That Heals Itself

```
if (!ubidots.connected()) {          // Wi-Fi drops, brokers restart
  ubidots.reconnect();
  ubidots.subscribeLastValue(DEVICE_LABEL, VAR_LED);  // easy to forget after a reconnect
}
if (millis() - lastPublish > PUBLISH_EVERY_MS) {
  if (!isnan(t) && !isnan(h)) {       // skip a bad reading: an honest gap beats a lie
    ubidots.add("temperature", t); ubidots.add("humidity", h);
    ubidots.publish(DEVICE_LABEL);    // both readings in one message
  }
  lastPublish = millis();
}
ubidots.loop();                       // lets incoming cloud messages be delivered
```

- The loop is now **closed**: device to cloud, and cloud to device. A device that also listens is a true IoT device.
- Network code must **self-heal**, timing must be **non-blocking** (`delay()` would stall `ubidots.loop()`), and the **token is a password**: keep it out of git.

The Reference Architecture 🗺️



Things 🔌 Devices and edge devices sensing the world, some running AI at the edge.

Insights ☁️ An **IoT service** ingests the data, then **serverless functions** and **storage** turn it into meaning.

Actions ⚡ Commands back to devices, plus **dashboards** and **analytics** for people.

Tip

PROJECT 12 · CLOUD / MQTT

ESP32 Telegram Bot

Chat with your board: read the sensor and switch the light from Telegram, anywhere.

Builds on: Project 11's cloud connection, swapping a dashboard for a chat app as the interface.

⏱️ ~35 min 📊 Intermediate

🔖 Prerequisite: Project 9 & 11

⬇ Download the sketch

Save `Project_12_ESP32_Telegram_Bot.ino`. Needs **UniversalTelegramBot**, **ArduinoJson** (v6+), and the **DHT** libraries, plus your Wi-Fi and a bot token.

Project 12 · Telegram Bot

🤖

Read the sensor and switch the light by chat, from anywhere.

- Use an existing chat app as the device's whole user interface
- Poll an outside service to reach the board on any network
- Parse a text command and dispatch to the right action
- Send an unprompted message when a sensor event happens

Key pins: DHT11 `4`, output `26`, PIR `27`

Your ESP32 in a Chat App

Control the board and read its sensor straight from **Telegram**, from anywhere. No page to design, nothing to install.

- `/status` returns temperature, humidity, and the light state
- `/light_on` and `/light_off` switch the LED or relay, `/help` lists commands
- Set your chat id and the board **messages you** when the PIR sees motion

`You (Telegram app)` ← HTTPS → `Telegram servers` ← HTTPS → `ESP32`

Because it is just **outbound** HTTPS, this reaches your phone even on networks where hosting a web server (Projects 5 to 9) never would.

HTTPS Needs Someone to Trust

```
WiFiClientSecure secured_client;                // TCP + TLS: the "s" in https
UniversalTelegramBot bot(BOT_TOKEN, secured_client);

// in setup():
secured_client.setCACert(TELEGRAM_CERTIFICATE_ROOT);  // trust Telegram's server
```

- A laptop ships with hundreds of **root certificates**; an ESP32 ships with **none**, so the library bundles the one Telegram chains up to and this line installs it. Skip it and every request fails, even with a valid token.
- `UniversalTelegramBot` wraps the API (turning `sendMessage()` into the right HTTPS request); **ArduinoJson** decodes the replies underneath, which is why it is a dependency.
- The **token** identifies your bot: keep it in the edit block, never public.

Poll for Commands, Push on Motion

```
if (motion && !lastMotion && String(CHAT_ID).length() > 0)
  bot.sendMessage(CHAT_ID, "Motion detected!", "");  // push, edge-detected
lastMotion = motion;

if (millis() - lastBotCheck > BOT_CHECK_INTERVAL) {   // poll about once a second
  int n = bot.getUpdates(bot.last_message_received + 1);   // "+1": newer than last seen
  while (n) { handleNewMessages(n); n = bot.getUpdates(bot.last_message_received + 1); }
  lastBotCheck = millis();
}
```

- The board always connects **outward**, which is why it works behind a router that would block an incoming request.
- `+ 1` asks only for messages **newer than the last handled**, so the bot never replays its own history.
- `handleNewMessages()` is a **command dispatcher**: compare `text` to each command, reply to that message's `chat_id`, and answer unknowns with "send /help".

Three Control Models

You have now built all three. The capstone is where you pick.

Model	Reach	Strength
Local web server (P5 to 9)	Same network only	Fast, private, no account
Cloud dashboard (P11)	Anywhere	History and charts
Chat bot (P12)	Anywhere	Instant, personal, push alerts

- The best UI is often **one people already have**: no app, no page, just `/help`
- **Outbound polling beats inbound hosting** on real-world networks
- Both directions matter: replies are **pull**, motion alerts are **push**
- Edge detection is not optional once a message **costs** something

Security in IoT 

*"The **S** in IoT stands for Security"*

Only as Secure as the Cloud

An IoT device connects to a cloud service, so it's **only as secure as that service**. Open connections invite malicious data and attacks, with **real-world** consequences.

Stuxnet worm

Manipulated **valves in centrifuges** to physically damage them: a cyber attack with kinetic consequences.

Hacked baby monitors

Poor passwords let attackers access home cameras and monitors.

Air-gapping

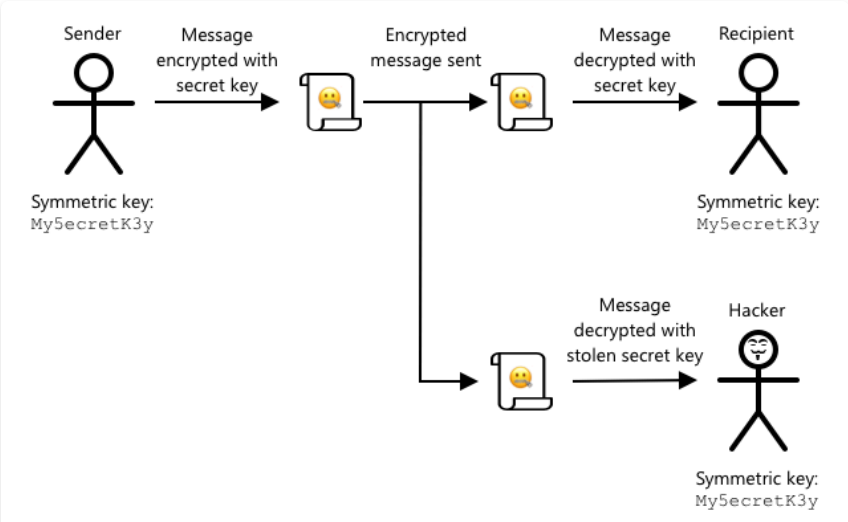
Some devices run on a network **completely isolated** from the Internet to stay private and secure.

Who Is Allowed to Connect?

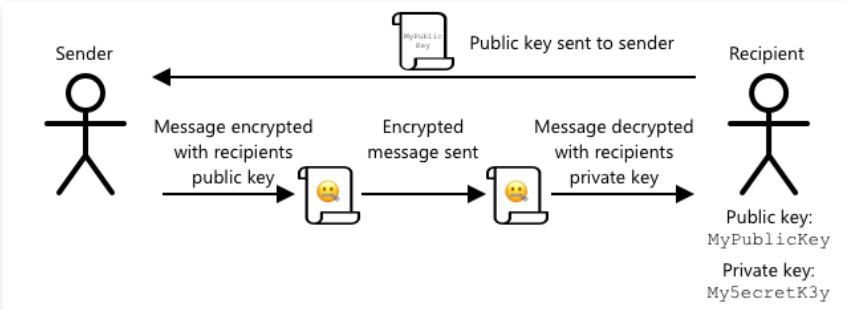
Every device gets its **own identity**: a secret key or certificate it presents when it connects. No valid credential, no connection.

> ⚠ **Warning**
>
> Per-device keys mean one compromised device can be `revoked` on its own, without locking out the whole fleet.

Keeping the Message Secret



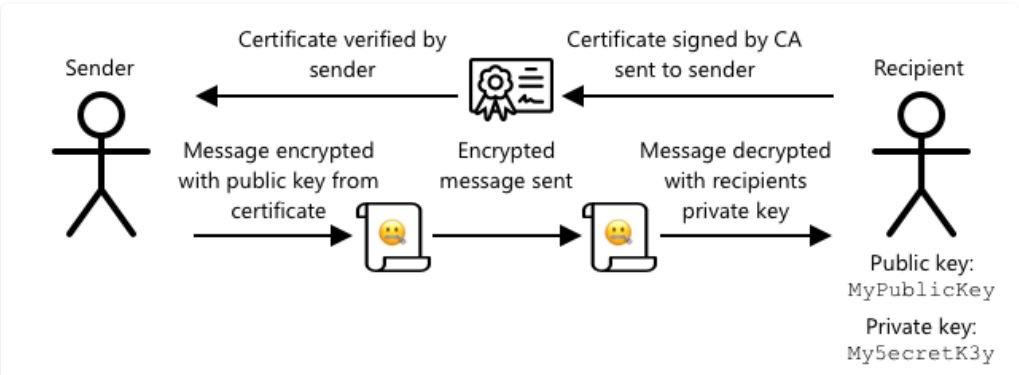
Symmetric: one shared key. Simple, but anyone who steals it can read everything.



Asymmetric: a **public** key encrypts, only the **private** key decrypts.

> **Note**
> Slower than symmetric, so real systems often use asymmetric keys just to agree on a symmetric one.

Certificates: Proving the Key Is Theirs



A **certificate** signed by a certificate authority proves a public key really belongs to who it claims to.

- The sender **verifies** it before trusting the key
- Devices can present certificates instead of secret keys
- The same idea behind the padlock in your browser

PROJECT 13 · CAPSTONE

Capstone: Build Your Own IoT Device

Combine the week into one device that senses, acts, and connects to the cloud, then demo it.

Builds on: everything so far. This is where the twelve separate projects stop being separate.

⏱️ ~60 min 📊 Advanced 🔖 Prerequisite: Projects 1-12

⬇ Download the starter sketch

The starter already connects to Wi-Fi and Ubidots. Follow the `TODO 1/2/3` markers to make it yours.

Project 13 · Capstone 🎓

Design and demo an IoT device of your own choosing, starter sketch and rubric included.

- Combine a sensor, an actuator, and a network service into one device
- Scope a build to the time available
- Start from a template and extend it, the way most embedded work begins
- Debug a multi-part system by testing one piece at a time

Key pins: your choice

Open the guide ↗

The Brief

In teams, combine the week into one small end-to-end device that does all three:

Sense
DHT11, PIR, LDR, pot

Act
LED, RGB, buzzer, relay, OLED

Connect
Ubidots and/or Telegram

Idea	Sense	Act	Cloud
Smart room monitor	DHT11 + LDR	OLED readings	Ubidots charts
Motion security node	PIR	Buzzer	Telegram alert
Comfort meter	DHT11	RGB green/yellow/red	Ubidots dashboard

One session to build, then a short demo. A simple thing that fully works beats an ambitious thing that does not.

Thank you!

For spending the day building the Internet of Things with us.

Questions? Ask now, or write to me any time.

Slides, labs, and resources

KGSP Summer Enrichment Program 2026
KAUST Academy, KAUST

Salah Abdeljabar
salah.abdeljabar@kaust.edu.sa