CS3237

Lecture 5

Backend Communications

colintan@nus.edu.sg

National University of Singapore

School of Computing

Motivation

- **We have been looking at various machine learning models:**
  - Statistical models
  - Simple dense neural networks
  - Deep learning models
- **Many of these models do not run well on edge devices; they need too much computing power.**
  - Edge devices need to send data to back-end cloud servers for further processing.
- **In this lecture we look at two options for you to do this, as well as two other related topics: Working with databases and setting passwords for MQTT.**
  - Hypertext Transfer Protocol (HTTP)
  - Message Queuing Telemetry Transport (MQTT)
  - Working with Databases
  - Securing MQTT with passwords.

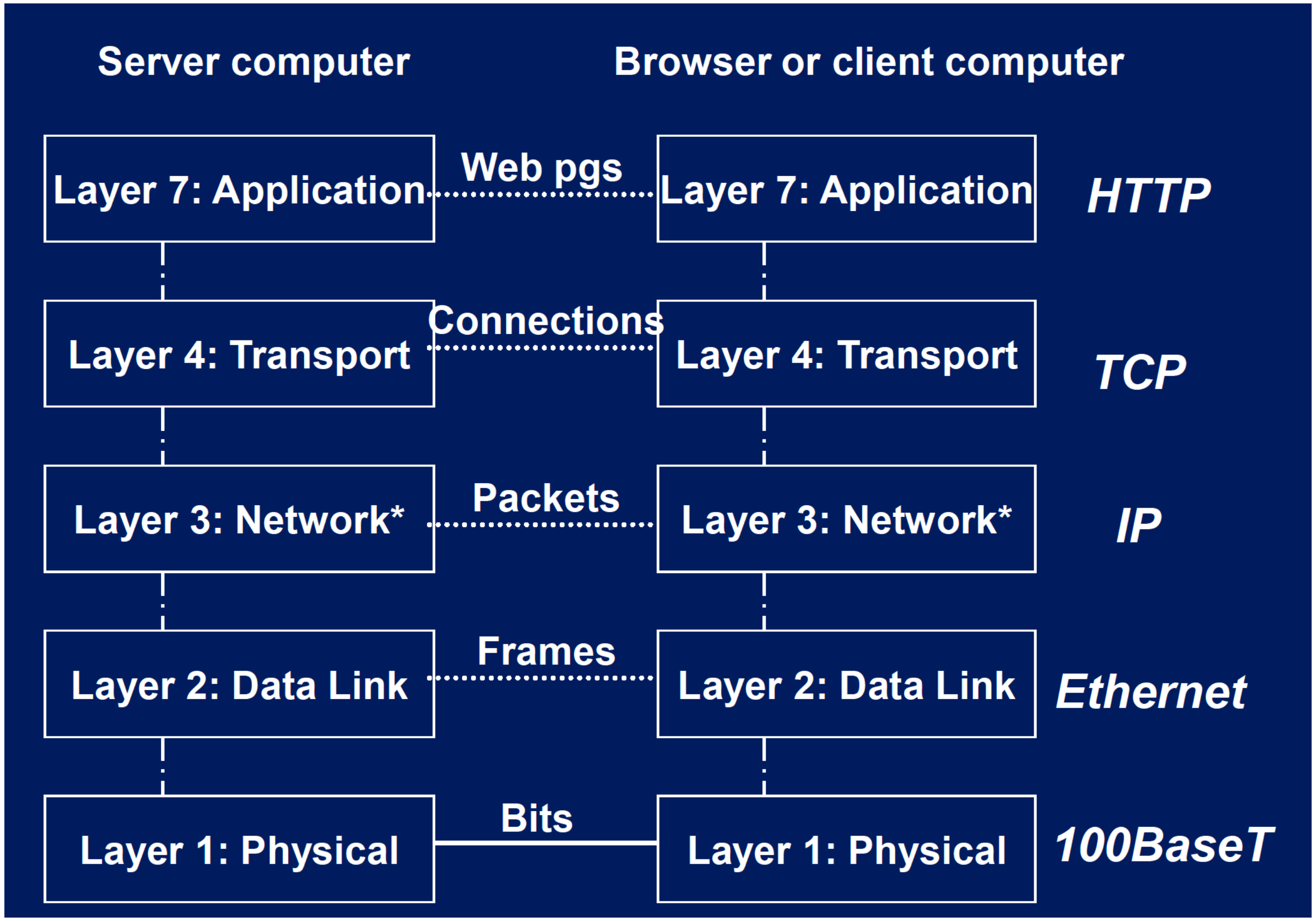
CS3237 Introduction to IoT

HYPERTEXT TRANSFER PROTOCOL

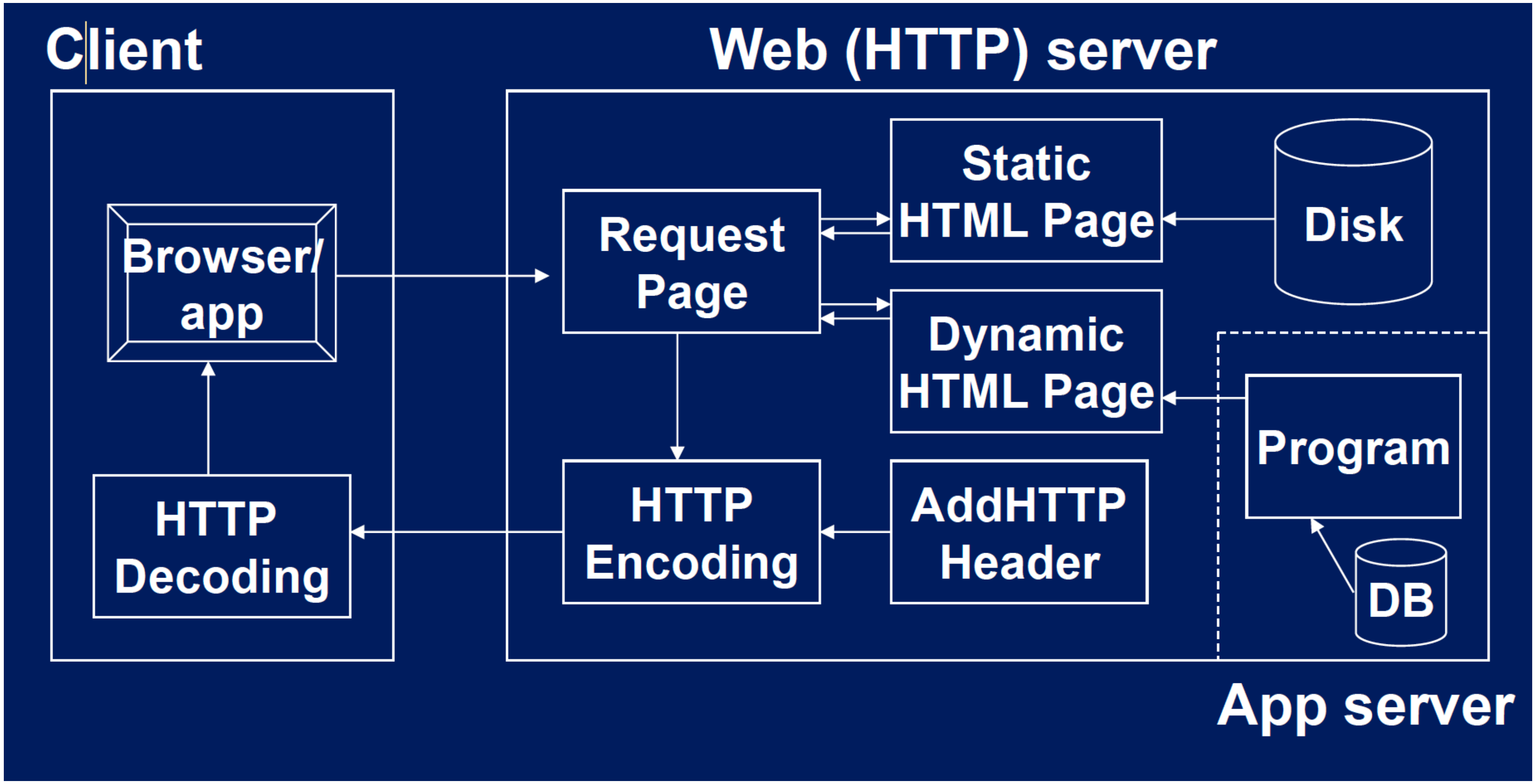
Hypertext Transfer Protocol (HTTP)

- **HTTP is the core request-response protocol for the web.**
  - Layer 7 (application layer) protocol.
- **Consists of four phases:**
  - Connection: Open a connection to the server.
  - Request: Make a request (GET, POST, PUT, DELETE).
  - Response: Receive a response from the server.
  - Close connection: Close the connection to the server.
- **HTTP is stateless**
  - HTTP does not keep track of what happened in previous connections.
  - Application backend must do this on its own through use of databases, etc.
  - Applications may also deposit “cookies” in the browser to maintain states.

Hypertext Transfer Protocol (HTTP)
OSI Model



Hypertext Transfer Protocol (HTTP)

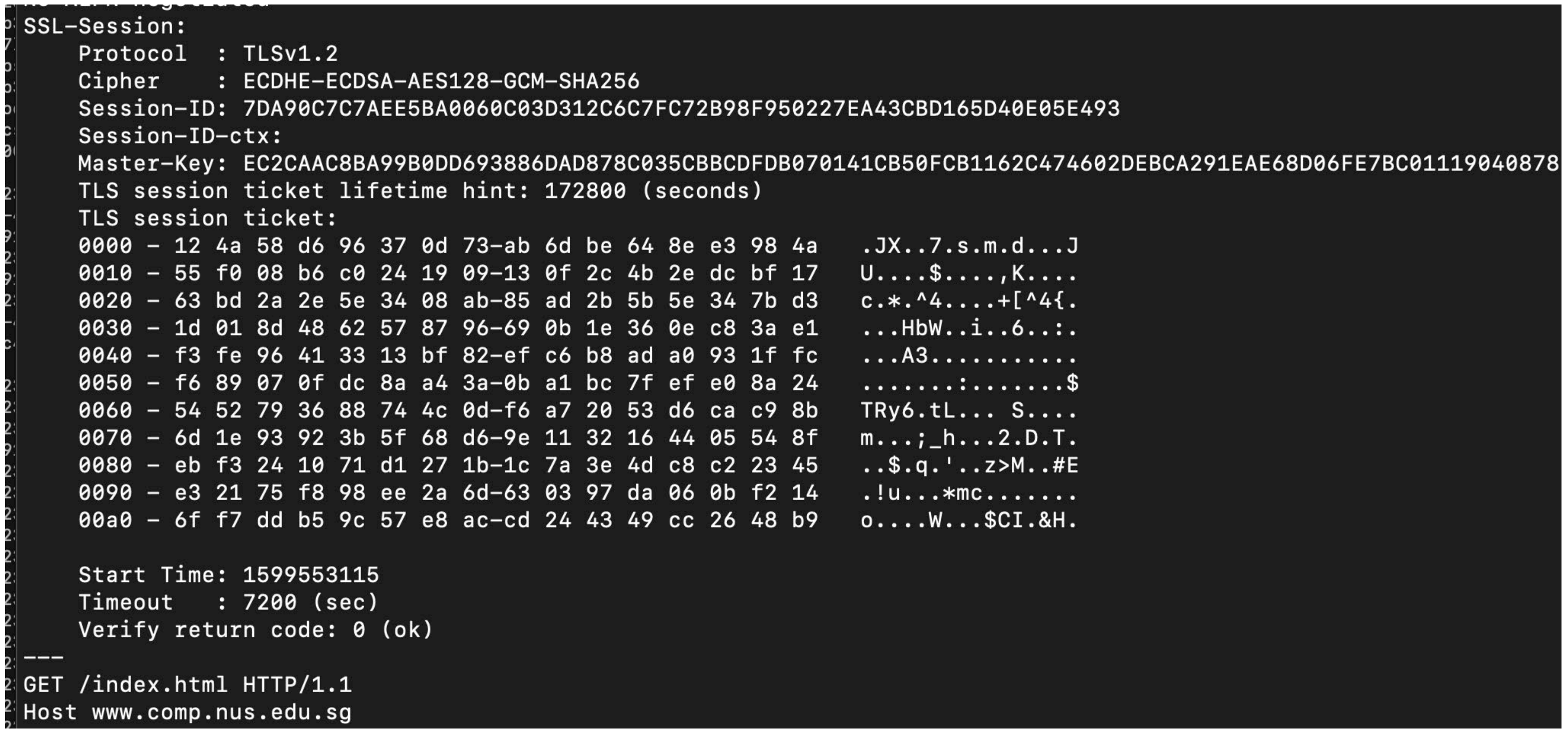


Hypertext Transfer Protocol (HTTP)

- **Example interaction:**

  openssl s_client –connect www.facebook.com:443

  <Connection information including certificate displayed>



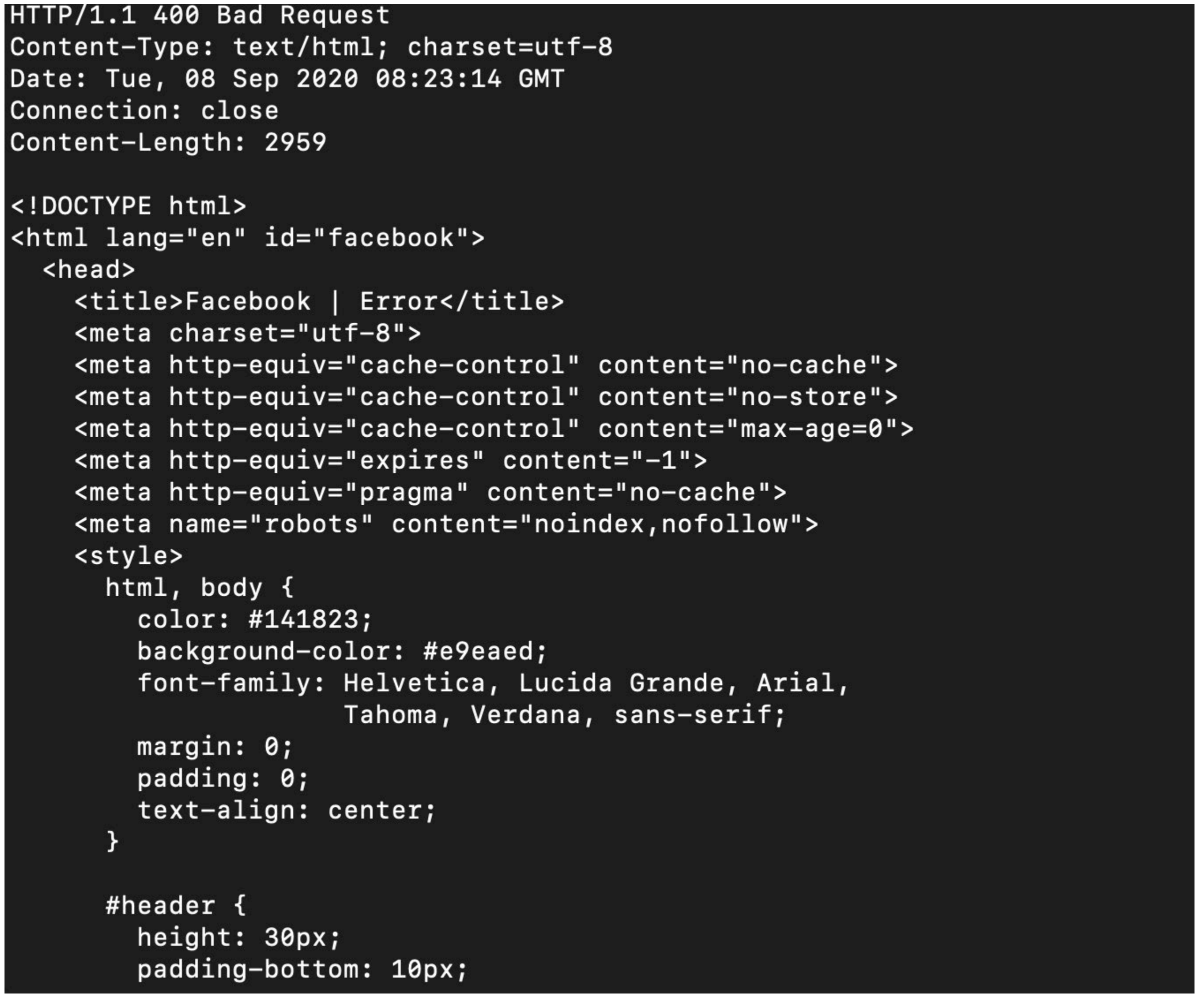
```
SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : ECDHE-ECDSA-AES128-GCM-SHA256
    Session-ID: 7DA90C7C7AEE5BA0060C03D312C6C7FC72B98F950227EA43CBD165D40E05E493
    Session-ID-ctx:
    Master-Key: EC2CAAC8BA99B0DD693886DAD878C035CBBCDFDB070141CB50FCB1162C474602DEBCA291EAE68D06FE7BC01119040878
    TLS session ticket lifetime hint: 172800 (seconds)
    TLS session ticket:
    0000 - 12 4a 58 d6 96 37 0d 73-ab 6d be 64 8e e3 98 4a   .JX..7.s.m.d...J
    0010 - 55 f0 08 b6 c0 24 19 09-13 0f 2c 4b 2e dc bf 17   U....$....,K....
    0020 - 63 bd 2a 2e 5e 34 08 ab-85 ad 2b 5b 5e 34 7b d3   c.*.^4....+[^4{.
    0030 - 1d 01 8d 48 62 57 87 96-69 0b 1e 36 0e c8 3a e1   ...HbW..i..6..:.
    0040 - f3 fe 96 41 33 13 bf 82-ef c6 b8 ad a0 93 1f fc   ...A3...........
    0050 - f6 89 07 0f dc 8a a4 3a-0b a1 bc 7f ef e0 8a 24   .......:.......$
    0060 - 54 52 79 36 88 74 4c 0d-f6 a7 20 53 d6 ca c9 8b   TRy6.tL... S....
    0070 - 6d 1e 93 92 3b 5f 68 d6-9e 11 32 16 44 05 54 8f   m...;_h...2.D.T.
    0080 - eb f3 24 10 71 d1 27 1b-1c 7a 3e 4d c8 c2 23 45   ..$.q.'..z>M..#E
    0090 - e3 21 75 f8 98 ee 2a 6d-63 03 97 da 06 0b f2 14   .!u...*mc.......
    00a0 - 6f f7 dd b5 9c 57 e8 ac-cd 24 43 49 cc 26 48 b9   o....W...$CI.&H.

    Start Time: 1599553115
    Timeout   : 7200 (sec)
    Verify return code: 0 (ok)
---
GET /index.html HTTP/1.1
Host www.comp.nus.edu.sg
```

Hypertext Transfer Protocol (HTTP)

- **Example interaction:**

  Response from server

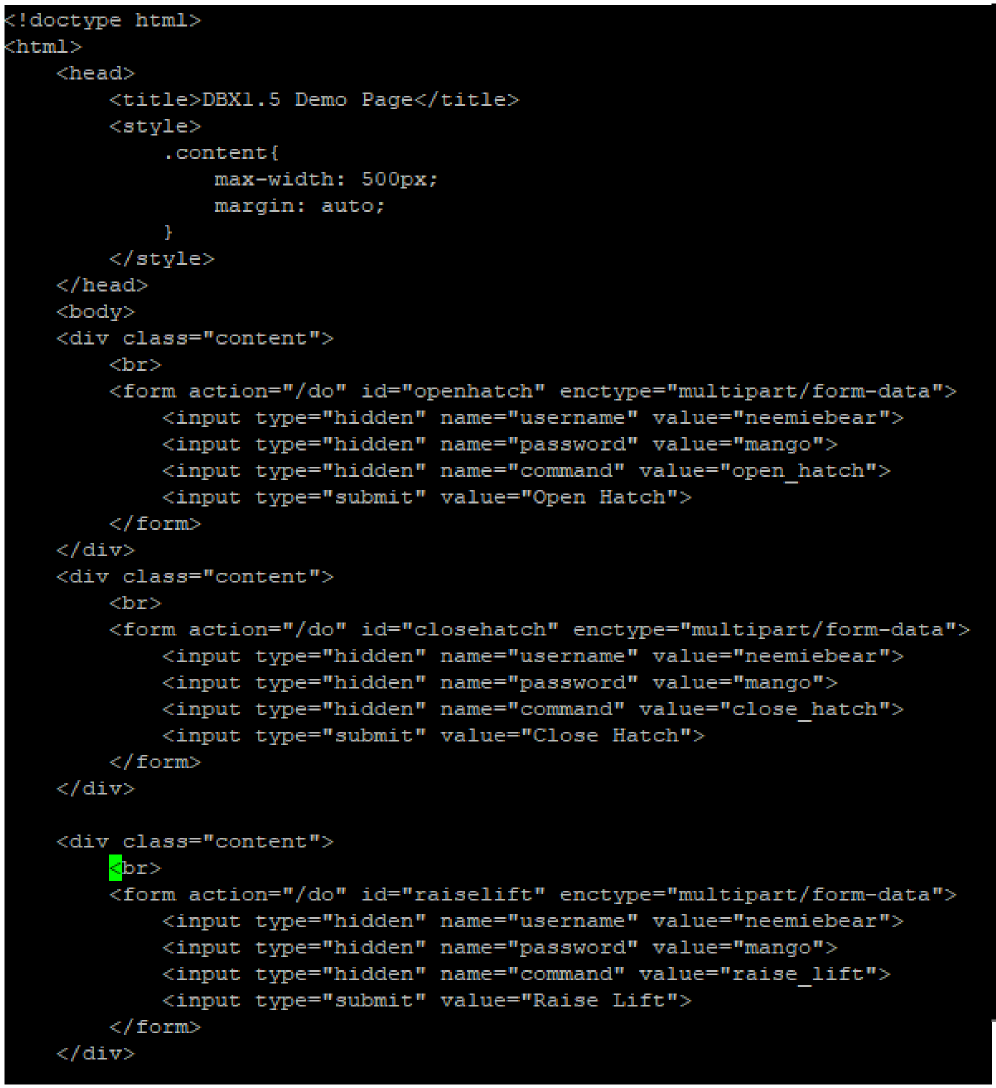
  (400: Bad Request. Oops!)



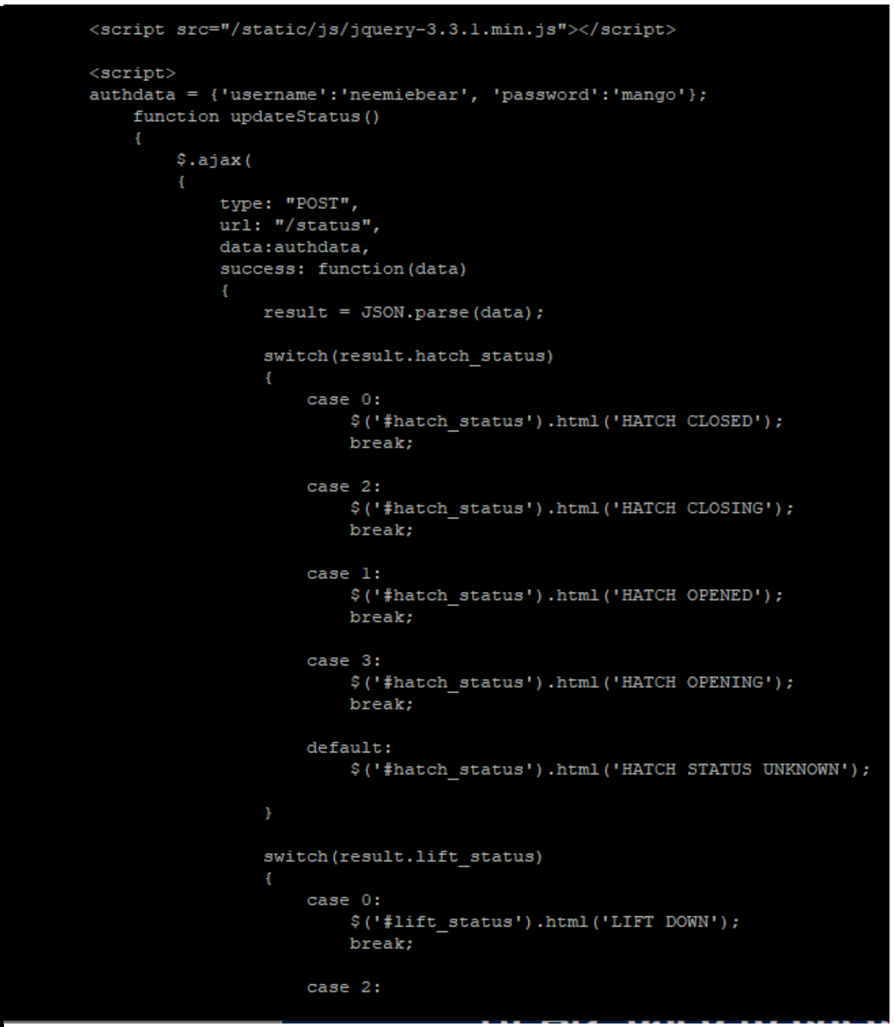
```
HTTP/1.1 400 Bad Request
Content-Type: text/html; charset=utf-8
Date: Tue, 08 Sep 2020 08:23:14 GMT
Connection: close
Content-Length: 2959

<!DOCTYPE html>
<html lang="en" id="facebook">
  <head>
    <title>Facebook | Error</title>
    <meta charset="utf-8">
    <meta http-equiv="cache-control" content="no-cache">
    <meta http-equiv="cache-control" content="no-store">
    <meta http-equiv="cache-control" content="max-age=0">
    <meta http-equiv="expires" content="-1">
    <meta http-equiv="pragma" content="no-cache">
    <meta name="robots" content="noindex,nofollow">
    <style>
      html, body {
        color: #141823;
        background-color: #e9eaed;
        font-family: Helvetica, Lucida Grande, Arial,
                     Tahoma, Verdana, sans-serif;
        margin: 0;
        padding: 0;
        text-align: center;
      }

      #header {
        height: 30px;
        padding-bottom: 10px;
```

HTTP Request Types

- **There are 5 more common request types (of which only GET and POST are used often):**
  - GET: Request for a specific document (Can also be used to send data)
  - POST: Request for server to accept data from browser
  - PUT: Replace a document with the data provided by the browser.
  - DELETE: Remove a document
  - HEAD: Retrieve only the header of a document
- **Documents are in the form of Hypertext Markup Language (HTML) files:**
  - Consists of:
    - ✓**Text that can be "marked-up" to format it (e.g. headings, tables, bold, italic, alignment, etc.)**
    - ✓**Hyperlinks: Links to other documents, possibly on other servers.**
    - ✓**Scripts: Programs that will be run on your browser.**



```
<!doctype html>
<html>
    <head>
        <title>DBX1.5 Demo Page</title>
        <style>
            .content{
                max-width: 500px;
                margin: auto;
            }
        </style>
    </head>
    <body>
    <div class="content">
        <br>
        <form action="/do" id="openhatch" enctype="multipart/form-data">
            <input type="hidden" name="username" value="neemiebear">
            <input type="hidden" name="password" value="mango">
            <input type="hidden" name="command" value="open_hatch">
            <input type="submit" value="Open Hatch">
        </form>
    </div>
    <div class="content">
        <br>
        <form action="/do" id="closehatch" enctype="multipart/form-data">
            <input type="hidden" name="username" value="neemiebear">
            <input type="hidden" name="password" value="mango">
            <input type="hidden" name="command" value="close_hatch">
            <input type="submit" value="Close Hatch">
        </form>
    </div>

    <div class="content">
        <br>
        <form action="/do" id="raiselift" enctype="multipart/form-data">
            <input type="hidden" name="username" value="neemiebear">
            <input type="hidden" name="password" value="mango">
            <input type="hidden" name="command" value="raise_lift">
            <input type="submit" value="Raise Lift">
        </form>
    </div>
```



```
<script src="/static/js/jquery-3.3.1.min.js"></script>

<script>
authdata = {'username':'neemiebear', 'password':'mango'};
    function updateStatus()
    {
        $.ajax(
        {
            type: "POST",
            url: "/status",
            data:authdata,
            success: function(data)
            {
                result = JSON.parse(data);

                switch(result.hatch_status)
                {
                    case 0:
                        $('#hatch_status').html('HATCH CLOSED');
                        break;

                    case 2:
                        $('#hatch_status').html('HATCH CLOSING');
                        break;

                    case 1:
                        $('#hatch_status').html('HATCH OPENED');
                        break;

                    case 3:
                        $('#hatch_status').html('HATCH OPENING');
                        break;

                    default:
                        $('#hatch_status').html('HATCH STATUS UNKNOWN');

                }

                switch(result.lift_status)
                {
                    case 0:
                        $('#lift_status').html('LIFT DOWN');
                        break;

                    case 2:
```

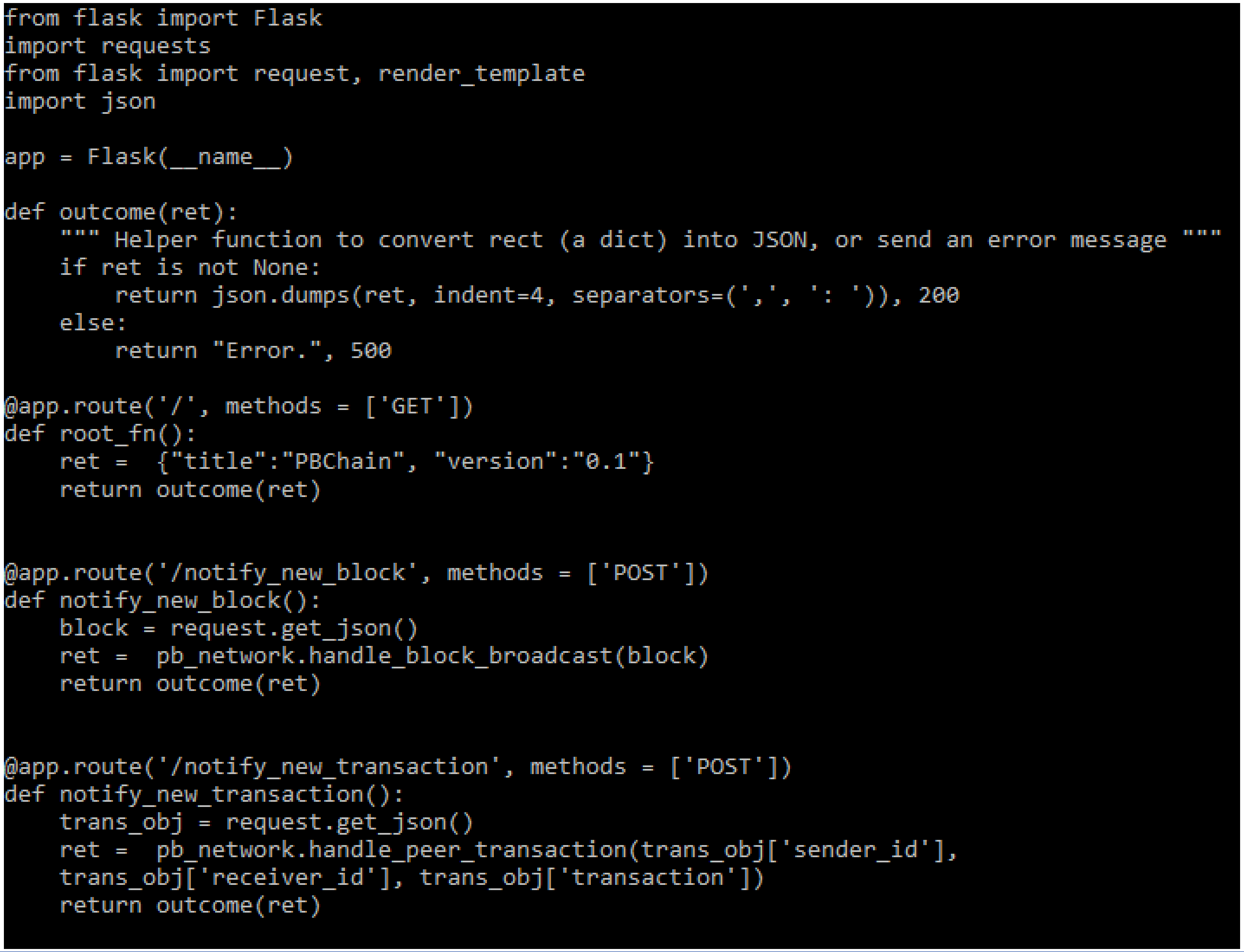
MIME Types

- **Documents have types:**
  - Multipurpose Internet Mail Extension (MIME) types:
    - ✓**Extensible. Can define new MIME types.**
    - ✓**Common MIME types:**
      - –*application/pdf*
      - –*application/json*
      - –*image/gif*
      - –*image/jpg*
      - –*text/html*
      - –*test/plain*
      - –*video/mpeg*
    - ✓**Browser will prompt if it sees an unexpected MIME type.**
  - When you request for a document using GET or POST, you would specify the expected document type in the header. For example:
    - ✓**‘Content-Type: application/json’**

Creating a HTTP Server in Python

- **Python has several frameworks for you to create HTTP servers:**
  - http.server within the Python library itself.
  - Flask, a very nice framework.
  - Django, another popular framework.
- **Flask and Django are full-stack frameworks:**
  - Provides not just HTTP, but also a way of incorporating HTML files.
- **Installing Flask:**
  - pip3 install flask

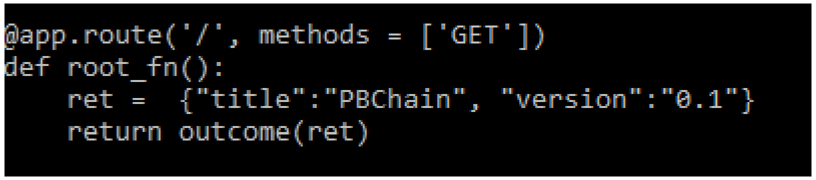
Example Flask Code



```python
from flask import Flask
import requests
from flask import request, render_template
import json

app = Flask(__name__)

def outcome(ret):
    """ Helper function to convert rect (a dict) into JSON, or send an error message """
    if ret is not None:
        return json.dumps(ret, indent=4, separators=(',', ': ')), 200
    else:
        return "Error.", 500

@app.route('/', methods = ['GET'])
def root_fn():
    ret =  {"title":"PBChain", "version":"0.1"}
    return outcome(ret)


@app.route('/notify_new_block', methods = ['POST'])
def notify_new_block():
    block = request.get_json()
    ret =  pb_network.handle_block_broadcast(block)
    return outcome(ret)


@app.route('/notify_new_transaction', methods = ['POST'])
def notify_new_transaction():
    trans_obj = request.get_json()
    ret =  pb_network.handle_peer_transaction(trans_obj['sender_id'],
    trans_obj['receiver_id'], trans_obj['transaction'])
    return outcome(ret)
```

Writing Flask Applications

- **Endpoints (e.g. /get_chain in http://localhost:5001/get_chain) are specified using the @app.route decorator, with the supported methods shown as a list in the "methods" parameter:**



```
@app.route('/', methods = ['GET'])
def root_fn():
    ret =  {"title":"PBChain", "version":"0.1"}
    return outcome(ret)
```

  - The function should return a string followed by a numeric result:
    - ✓**return "Success", 200**
  - Common return codes:
    - ✓**200: Success**
    - ✓**400: Request error**
    - ✓**404: Not found**
    - ✓**500: Server error**
- **The server can be started using "app.run()"**

Writing Flask Applications

- **HTML pages can be incorporated by calling "render_template" with a path to the template file (usually in the "./templates" directory), and parameters to pass to the template:**

```
return render_template('info.html', host = pb_host + ':' + str(pb_port),
info = info)
```

- **Templates in Flask are written using the Jinja scripting language (https://jinja.palletsprojects.com/en/2.11.x/) , supporting loops, parameter passing, etc.**

```
<html>
    <head>
        <title>INFORMATION PANEL FOR {{host}}</title>
    </head>
    <body>
        <center><h1>INFORMATION PANEL FOR {{host}}</h1></center>
        <h2>COIN BALANCES</h2>
        {% for balance in info.balances %}
        <p>{{balance.ID}}: {{balance.balance}} coins</p>
        {% endfor %}
        <h2>COIN HOLD BALANCES</h2>
        {% for balance in info.hold_balances %}
        <p>{{balance.ID}}: {{balance.hold_balance}} coins</p>
        {% endfor %}
        <h2>UNCONFIRMED TRANSACTIONS</h2>
        {% for transaction in info.unconfirmed %}
        <p>{{transaction.transaction}}</p>
        {% endfor %}
        <br>
    </body>
</html>
```

Writing Flask Applications

- **See ‘testsite.ipynb’ for example Flask website.**

Flask References

- **Amazing tutorial here on Flask (including incorporating MySQL databases):** https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world
- **Flask Reference:** https://flask.palletsprojects.com/en/1.1.x/

Creating HTTP Requests

- **You can create HTTP requests in Python by using the requests class.**
- **See 'writesite.ipynb' for an example, created to work with the 'testsite.ipynb' notebook.**

CS3237 Introduction to IoT

MESSAGE QUEUING TELEMETRY TRANSPORT

MQTT

- **Message Queuing Telemetry Transport (MQTT) is an ISO standard, lightweight protocol for transferring messages across IoT devices.**
- **It has several important advantages over HTTP:**
  - Lightweight, less overheads (don't need to specify document type, simpler headers).
  - Publish-subscribe model:
    - ✓**Centralized server.**
    - ✓**Clients subscribe to topics.**
    - ✓**When someone publishes messages in a topic, all interested clients are notified.**
  - Contrast with HTML:
    - ✓**Essentially a point-to-point (client to server) model.**
    - ✓**Client needs to continually poll the server using a client-side script, to get updates.**
    - ✓**Some "improvements" are available to solve this, e.g. websockets.**
- **We will have a lab for MQTT (but not for HTTP), so we will keep this lecture short.**

MQTT

- **To implement MQTT, you need a central server (called a "broker") that all clients will connect to:**
  - Recommended: Mosquitto. We will also install the Paho-MQTT client for Python.
  - LINUX:

    ```
    sudo apt-get install mosquitto mosquitto-clients
    pip3 install paho-mqtt
    ```

  - MacOS (Mosquitto client is automatically installed):
    - ✓**brew install mosquitto**
    - ✓**pip3 install paho-mqtt**
  - Windows:
    - ✓**Install and run the Windows Subsystem for LINUX. Instructions: https://docs.microsoft.com/en-us/windows/wsl/install-win10 and follow the instructions above for LINUX.**

Playing with MQTT

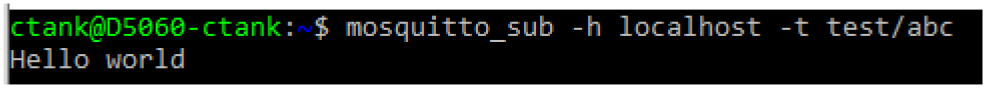
- **Once you have Mosquitto and the Mosquitto clients installed:**
  - LINUX: Open a new shell and type "mosquitto" to start the server
    - ✓ **sudo service mosquitto start**
  - MacOS: Starts automatically, otherwise:
    - ✓ **brew services start mosquitto**
- **MQTT messages are published with a "topic/subtopic" tag.**
  - Examples:
    - ✓ **messages**
    - ✓ **messages/sensor**
    - ✓ **messages/sensor/sensor1**
    - ✓ **etc.**

Playing with MQTT

- **Start two more LINUX / MacOS terminals:**
  - In one we will use mosquitto_sub to subscribe to messages published under the topic test/abc

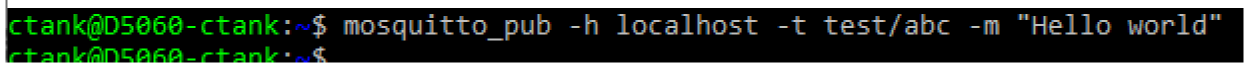
    mosquitto_sub –h localhost –t test/abc
  - In the other, we will use mosquitt_pub to publish a message:

    mosquitto_pub –h localhost –t test/abc –m “Hello world”
- **Results:**
  - Subscribe side (Waits until you publish, then prints message and waits again):



```
ctank@D5060-ctank:~$ mosquitto_sub -h localhost -t test/abc
Hello world
```

  - Publish side (Nothing happens, returns immediately):



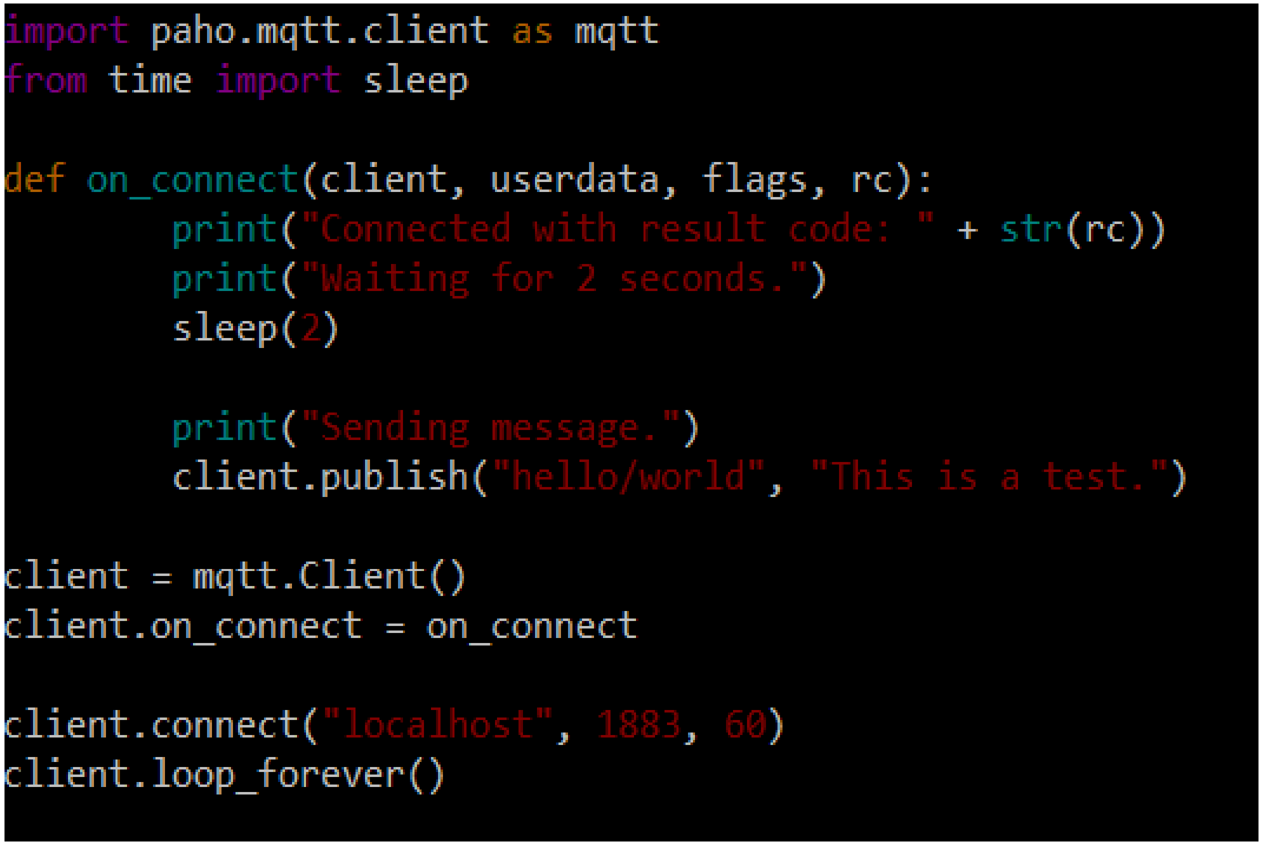
```
ctank@D5060-ctank:~$ mosquitto_pub -h localhost -t test/abc -m "Hello world"
ctank@D5060-ctank:~$
```

MQTT Programming in Python

- **We will use Paho-MQTT to publish/subscribe to a broker.**
  - Note that publisher/subscriber relationships in MQTT are symmetric:
    - ✓ **A publisher can be a subscriber and vice-versa**
  - Paho-MQTT is an event-driven library:
    - ✓ **We declare two listeners for:**
      - *on_connect: Called when the client has attempted to connect to a broker. Must get a result code of 0 for success.*
      - *on_message: Called when the client has received a message on the topic it is subscribing to.*
  - Example sender code is given on the next slide (mqtt_send.py)
    - ✓ **We call "connect" to connect to the broker, and "loop_forever" to get the event loop going.**

MQTT Programming in Python

Example sender code (mqtt_send.py)



```python
import paho.mqtt.client as mqtt
from time import sleep

def on_connect(client, userdata, flags, rc):
        print("Connected with result code: " + str(rc))
        print("Waiting for 2 seconds.")
        sleep(2)

        print("Sending message.")
        client.publish("hello/world", "This is a test.")

client = mqtt.Client()
client.on_connect = on_connect

client.connect("localhost", 1883, 60)
client.loop_forever()
```

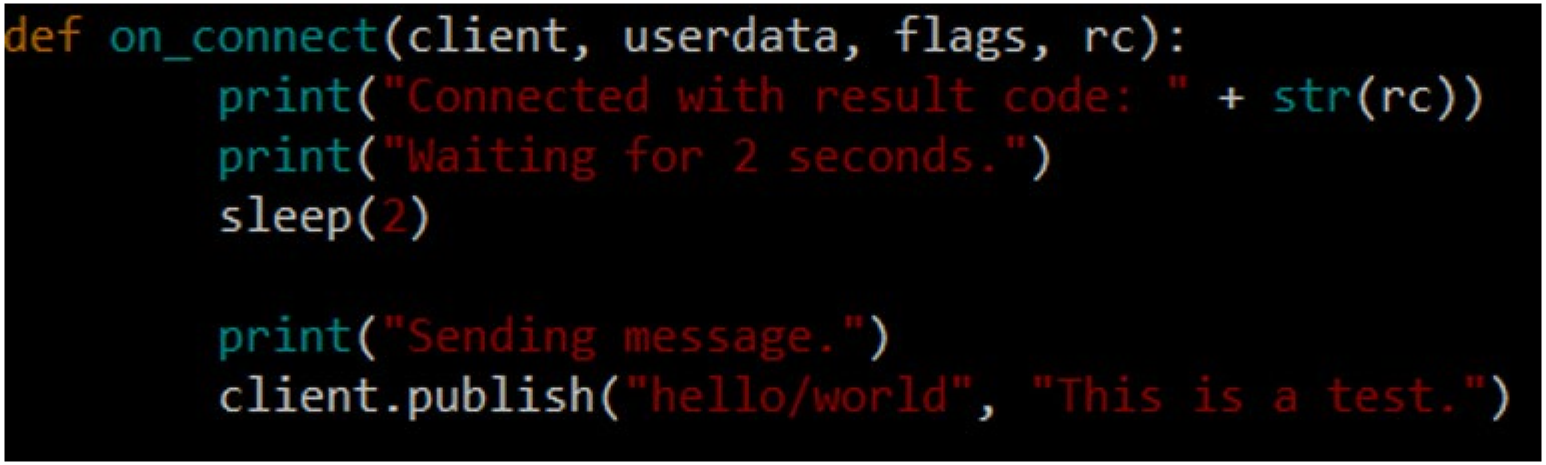
MQTT Programming in Python

- **The table below shows the parameters for on_connect and their meanings:**

Parameter	Description
client	Instance of the Paho MQTT client library.
userdata	Private user data that can be set in the MQTT Client() constructor. Not used here.
flags	A dictionary containing flags returned by the broker. Not used here.
rc	Result code: 0 – OK 1 – Connection refused, incorrect protocol version. 2 – Connection refused, invalid client identifier. 3 – Connection refused, service not available. 4 – Connection refused, bad user name and password. 5 – Connection refused, not authorized.

MQTT Programming in Python

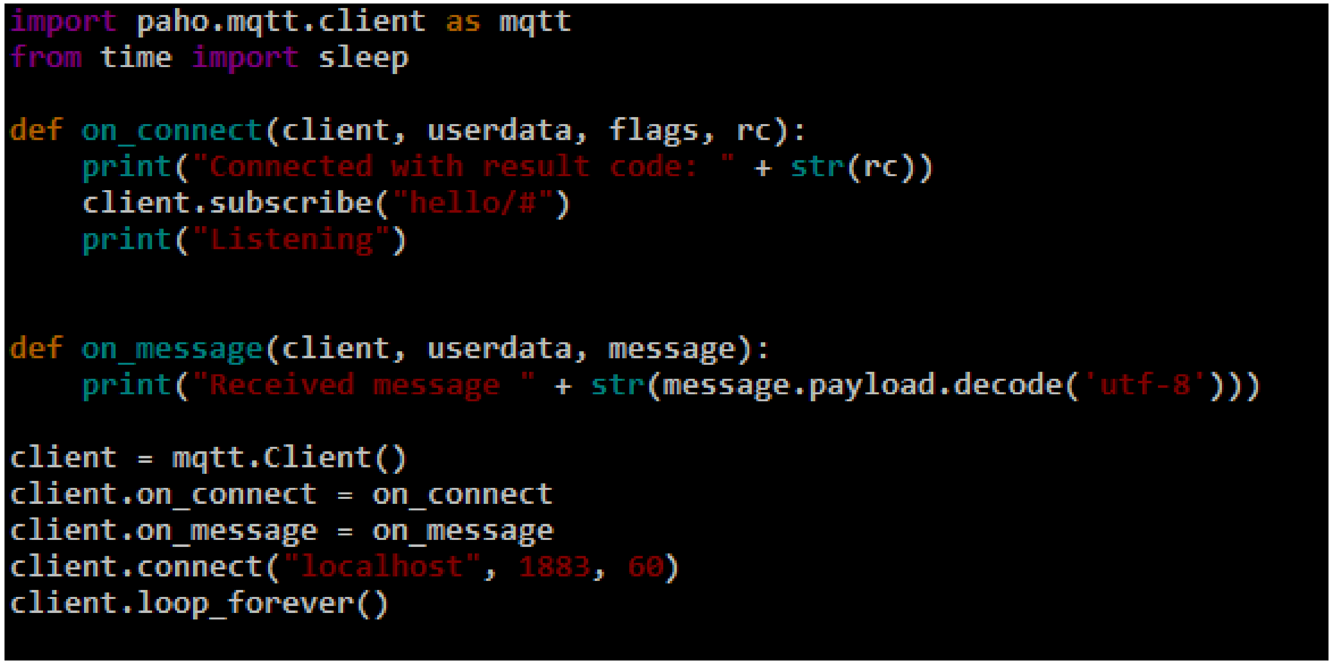
- **Publishing a message is easy:**
  - Call “publish” with the message and topic. We can modify our on_connect to do:



```
def on_connect(client, userdata, flags, rc):
        print("Connected with result code: " + str(rc))
        print("Waiting for 2 seconds.")
        sleep(2)

        print("Sending message.")
        client.publish("hello/world", "This is a test.")
```

MQTT Programming in Python

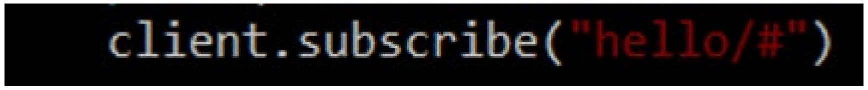
- **Example listener code (mqtt_listen.py):**



```
import paho.mqtt.client as mqtt
from time import sleep

def on_connect(client, userdata, flags, rc):
    print("Connected with result code: " + str(rc))
    client.subscribe("hello/#")
    print("Listening")


def on_message(client, userdata, message):
    print("Received message " + str(message.payload.decode('utf-8')))

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.connect("localhost", 1883, 60)
client.loop_forever()
```

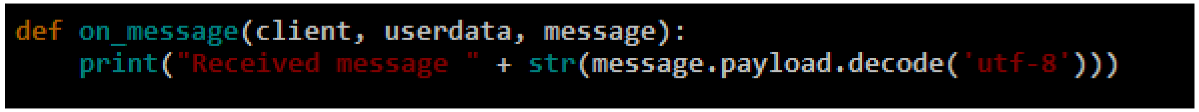
MQTT Programming in Python

- **Subscribing is easy (mqtt_listen.py):**
  - Call "subscribe" with the topic name. We can see it in the on_connect callback in mqtt_listen.py:



```
    client.subscribe("hello/#")
```

  - Here hello/# means listen to all subtopics under hello.
- **We receive messages using the on_listen callback:**



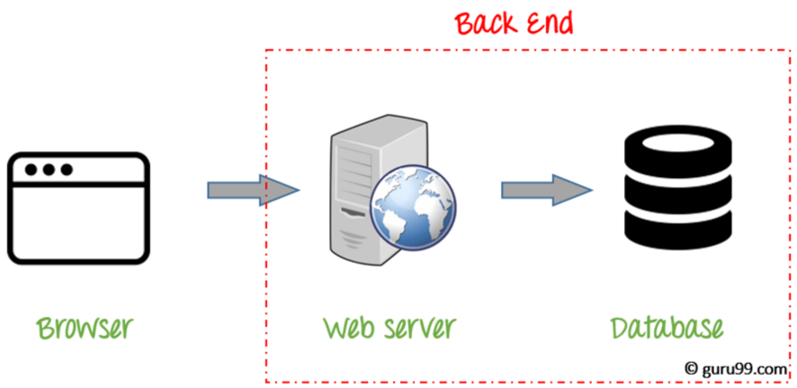
```
def on_message(client, userdata, message):
    print("Received message " + str(message.payload.decode('utf-8')))
```

CS3237 Introduction to IoT

WORKING WITH DATABASES

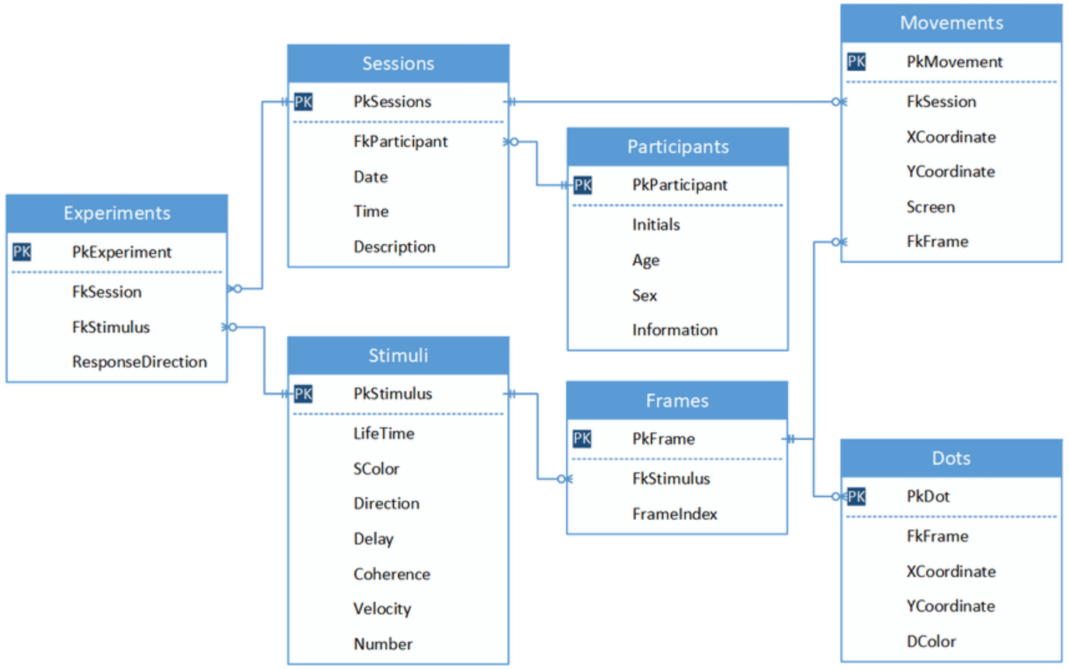
Working with Databases

- **Motivation:**
  - We can now transfer data between the client and the backend server.
  - How do we store this data? Two choices:
    - ✓**As flat files – Difficult to search.**
    - ✓**As a database – More complex solution, but easier to search for particular images or pieces of data. E.g. data read from a certain range of dates.**



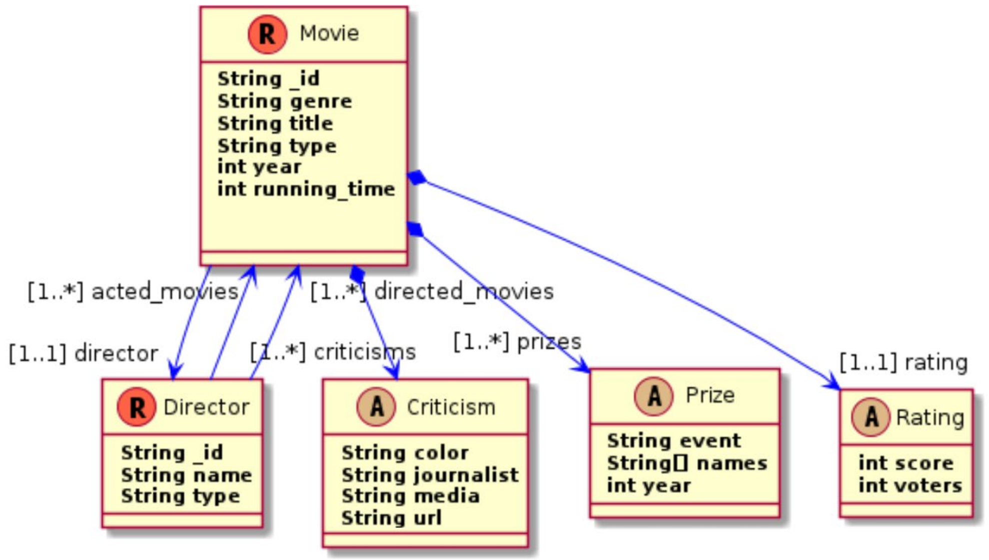
Working with Databases

- **There are two types of commonly available databases:**
  - Relational (e.g. MySQL):
    - ✓**Data is organized as tables.**
    - ✓**Data is indexed by "keys", especially the "primary key".**
      - *–Main identifying key for a piece of data.*
      - *–Must be unique.*
    - ✓**Relationships between tables are formed by declaring foreign keys into the keys of other tables:**
      - *–Tight relationships governed by rules to maintain integrity of database.*



Working with Databases

- Document Oriented (e.g. MongoDB)
  - ✓Organized as collections of documents, which may or may not be related.
  - ✓Only loose relationships are established between documents, or between collections of documents.
  - ✓Often called "NoSQL" for this reason.



Working with Databases

- **For the most part relationships won't be so useful for doing deep-learning and classification/regression on IoT data.**
  - We can store each image, reading etc. as separate documents.
  - We can still establish relationships between documents within our code.
- **We will focus only on MongoDB**
- **See https://docs.mongodb.com/manual/installation/ for how to install MongoDB.**

MongoDB Programming in Python

- **We will use PyMongo to read/write the MongoDB database:**

  pip3 install pymongo
- **Please see the “mongo.ipynb” notebook for programming examples.**

CS3237 Introduction to IoT

SECURING COMMUNICATIONS

Securing Communications - Motivation

- **IoT devices produce highly sensitive data:**
  - Smart bulb data can tell when you switch on the lights and when you switch them off.
    - ✓**Good clue to when you are in, when you are out, when you are awake, when you are asleep. Perfect for planning a robbery!**
  - Healthcare IoT data is highly confidential.
    - ✓**A rogue employer can use your healthcare data to fabricate a case for firing you without compensation.**
    - ✓**Insurance providers can use your data to hike your premiums or deny you coverage or claims.**
  - Industrial IoT sensors can provide data on material types, capacities, fabrication stages, etc.
    - ✓**A competitor can steal trade secrets without ever being inside the factory, just by listening to WiFi traffic.**
  - Attackers can control your smart devices remotely.
    - ✓**Turn on the microphone on your smart TV to listen to your conversations, etc.**

Securing Communications - Motivation

- **The MQTT server you have set up is highly vulnerable.**
  - Anyone who is on the same WiFi as you and knows your server address can intercept and send messages to your MQTT broker.
  - This means anyone can read your data, and activate your actuators just by being on the same WiFi as you and knowing your broker IP address.
- **There are two ways to secure your server:**
  - Using access control lists to control who can send and receive messages.
  - Using Transport Layer Security (TLS) to encrypt data and to validate clients. This will not be covered today.

Securing MQTT
Create Usernames and Passwords

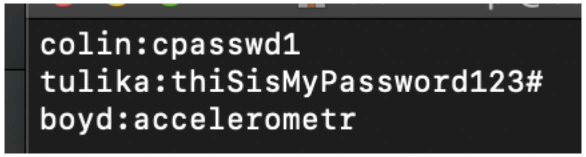
- **The first level of securing your MQTT broker is to create usernames and passwords.**
  - Create a file called "users.txt" with the following information on each line:

  <username1>:<password1>

  <username2>:<password2>

  …

  You can see a sample file here with three users and passwords:



```
colin:cpasswd1
tulika:thiSisMyPassword123#
boyd:accelerometr
```

Securing MQTT
Create Usernames and Passwords

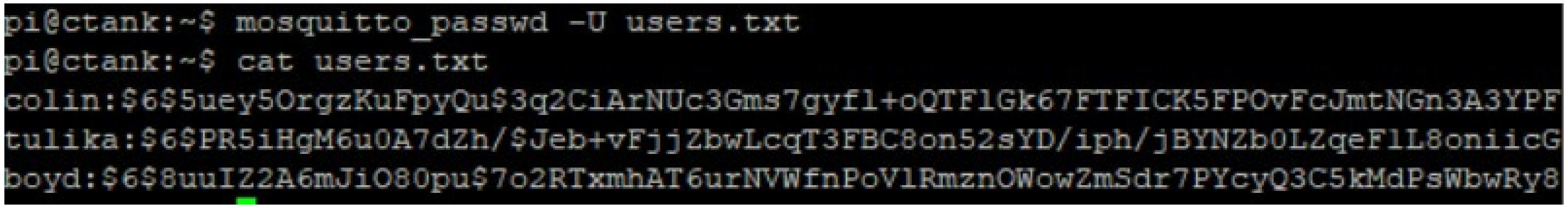
- **You can now create the password file:**

  mosquitto_passwd –U <password file>

  ▪In our case <password file> is user.txt, so we have:

  mosquitto_passwd –U users.txt

  ▪This produces an encrypted user file, as shown (Note: Original users.txt file is overwritten, so back this up if you need to):



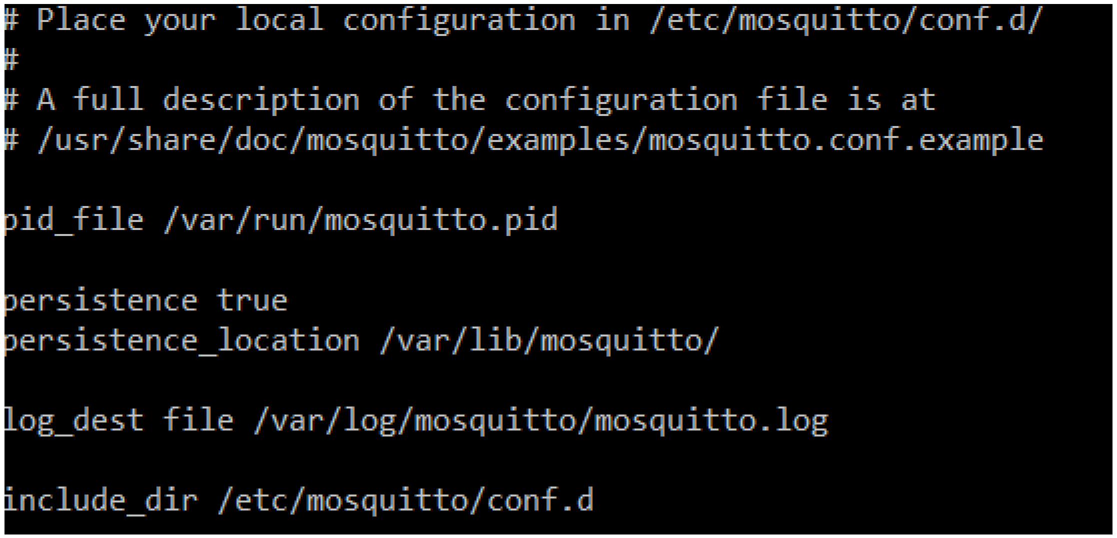
```
pi@ctank:~$ mosquitto_passwd -U users.txt
pi@ctank:~$ cat users.txt
colin:$6$5uey5OrgzKuFpyQu$3q2CiArNUc3Gms7gyfl+oQTFlGk67FTFICK5FPOvFcJmtNGn3A3YPF
tulika:$6$PR5iHgM6u0A7dZh/$Jeb+vFjjZbwLcqT3FBC8on52sYD/iph/jBYNZb0LZqeFlL8oniicG
boyd:$6$8uuIZ2A6mJiO80pu$7o2RTxmhAT6urNVWfnPoV1RmznOWowZmSdr7PYcyQ3C5kMdPsWbwRy8
```

Securing MQTT
Create Usernames and Passwords

- **Now you need to edit your mosquitto.conf file to include the new password file. On Ubuntu and MacOS it is at /etc/mosquitto/mosquitto.conf. To edit:**

  sudo vim /etc/mosquitto/mosquitto.conf

  ▪ This will bring up:



```
# Place your local configuration in /etc/mosquitto/conf.d/
#
# A full description of the configuration file is at
# /usr/share/doc/mosquitto/examples/mosquitto.conf.example

pid_file /var/run/mosquitto.pid

persistence true
persistence_location /var/lib/mosquitto/

log_dest file /var/log/mosquitto/mosquitto.log

include_dir /etc/mosquitto/conf.d
```

Securing MQTT
Create Usernames and Passwords

- **Add the following lines:**

```
allow_anonymous false
password_file <path to password file>
```

▪If your password file is at /home/pi/mqtt/users.txt, then the above will be:

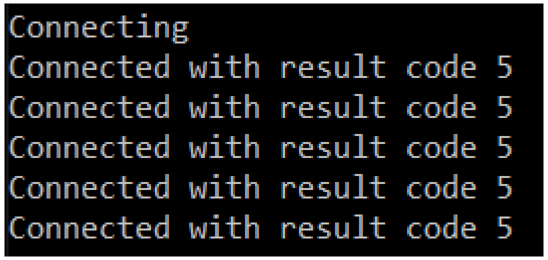
```
allow_anonymous false
password_file /home/pi/mqtt/users.txt
```

▪Note that you must provide the full path to the password file; you cannot use "~" to provide a relative path. Restart Mosquitto after making the changes:

```
sudo service mosquitto restart
```

Securing MQTT
Create Usernames and Passwords

- **Now if you connect to your broker using the mqtt.py program that you wrote in the lab, you will see:**



```
Connecting
Connected with result code 5
Connected with result code 5
Connected with result code 5
Connected with result code 5
Connected with result code 5
```

- **From our table of result codes in Lab 3, we see that result code 5 is "Connection refused, not authorized."**

Securing MQTT
Create Usernames and Passwords

- **You need to add in your username and password into your mqtt.py code. To do this we declare two constants USERID and PASSWORD and set these to the userid and password we had in users.txt:**

  USERID="colin"

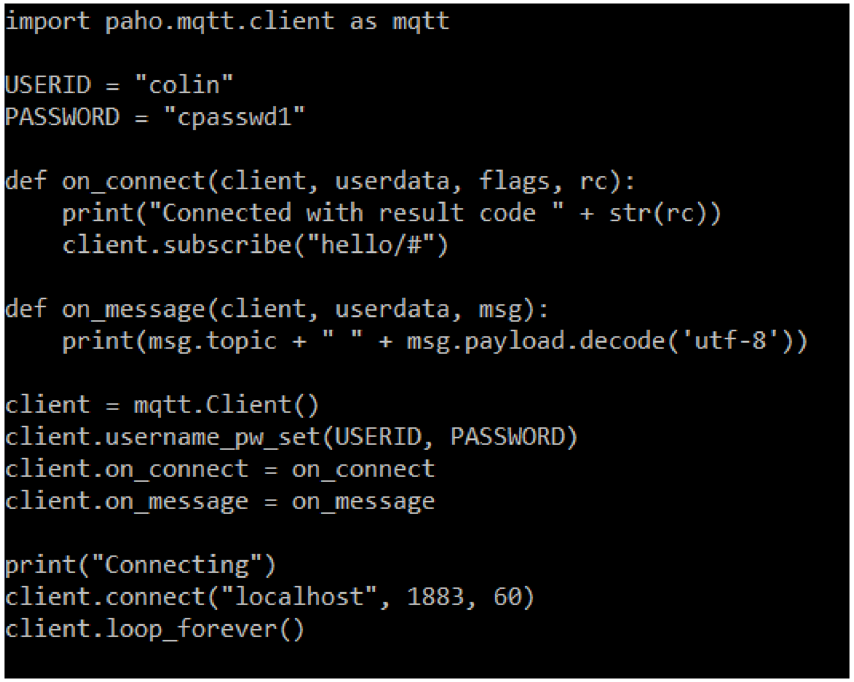
  PASSWORD="cpasswd1"

- **Now we add in the code to log in. Just after the line "client=mqtt.Client()" add in:**

  client.username_pw_set(USERID, PASSWORD)

Securing MQTT
Create Usernames and Passwords

- **Your code will now look like this:**



```
import paho.mqtt.client as mqtt

USERID = "colin"
PASSWORD = "cpasswd1"

def on_connect(client, userdata, flags, rc):
    print("Connected with result code " + str(rc))
    client.subscribe("hello/#")

def on_message(client, userdata, msg):
    print(msg.topic + " " + msg.payload.decode('utf-8'))

client = mqtt.Client()
client.username_pw_set(USERID, PASSWORD)
client.on_connect = on_connect
client.on_message = on_message

print("Connecting")
client.connect("localhost", 1883, 60)
client.loop_forever()
```

- **When you run the code you will see that it now connects with a result code of 0.**

Summary

- **In this lecture we looked at two alternatives for communications between nodes:**
  - MQTT – Best for edge devices to gateways.
  - HTTP – Best for gateways to the cloud
- **We also looked at databases:**
  - Two main types – Relational and Document
  - We looked at MongoDB.
- **Finally we looked at how to set passwords for MQTT.**