# UNIT – IV

**IoT Physical Devices & Endpoints:** Raspberry Pi, About the Board, Linux on Raspberry Pi, Raspberry Pi Interfaces, Programming Raspberry Pi with Python, Other IoT Devices, IoT Physical Servers & Cloud Offerings, Introduction to Cloud Storage Models & Communication APIs, WAMP - AutoBahn for IoT, Xively Cloud for IoT, Python Web Application Framework - Django, Designing a RESTful Web API, Amazon Web Services, SkyNet IoT Messaging Platform.

# IOT PHYSICAL DEVICES & ENDPOINTS

## IoT Device:

A "Thing" in Internet of Things (IoT) can be any object that has a unique identifier and which can send/receive data (including user data) over a network (e.g., smart phone, smartTV, computer, refrigerator, car, etc.).

IoT devices are connected to the Internet and send information about themselves or about their surroundings (e.g. information sensed by the connected sensors) over a network (to other devices or servers/storage) or allow actuation upon the physical entities/environment around them remotely.

**IoT Device Examples:**

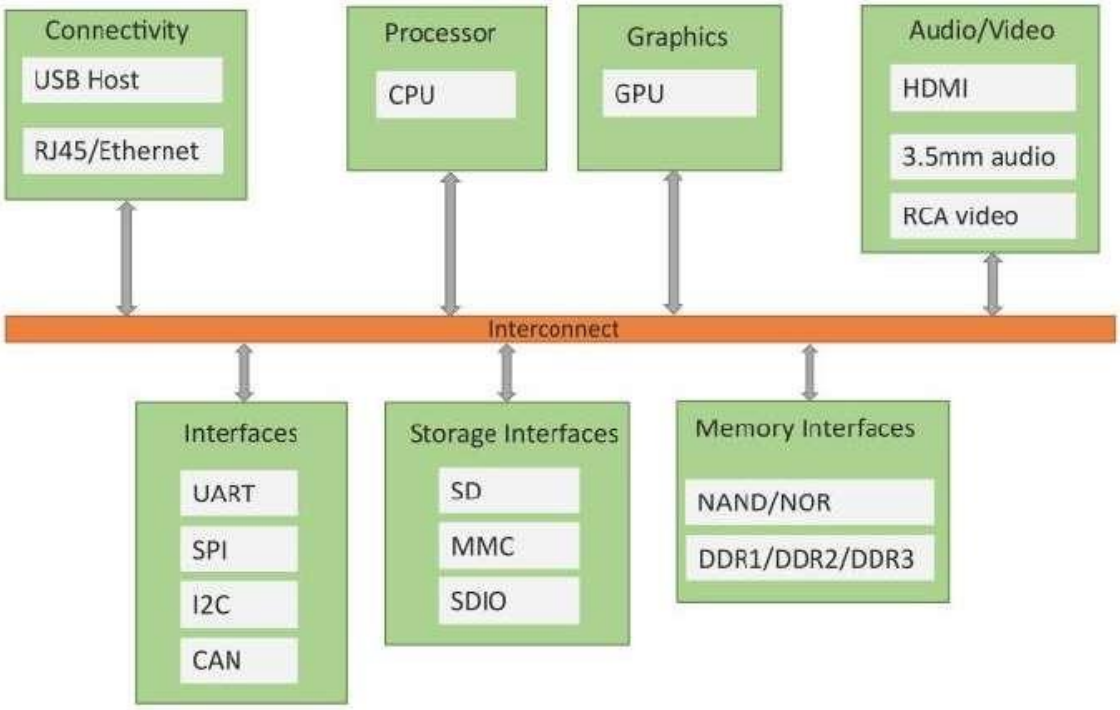
1. A home automation device that allows remotely monitoring the status of appliances and controlling the appliances.
2. An industrial machine which sends information abouts its operation and health monitoring data to a server.
3. A car which sends information about its location to a cloud-based service.
4. A wireless-enabled wearable device that measures data about a person such as the number of steps walked and sends the data to a cloud-based service.

**Basic building blocks of an IoT Device:**

1. **Sensing:** Sensors can be either on-board the IoT device or attached to the device.
2. **Actuation:** IoT devices can have various types of actuators attached that allow taking actions upon the physical entities in the vicinity of the device.
3. **Communication:** Communication modules are responsible for sending collected data to other devices or cloud-based servers/storage and receiving data from other devices and commands from remote applications.

4. **Analysis & Processing:** Analysis and processing modules are responsible for making sense of the collected data.

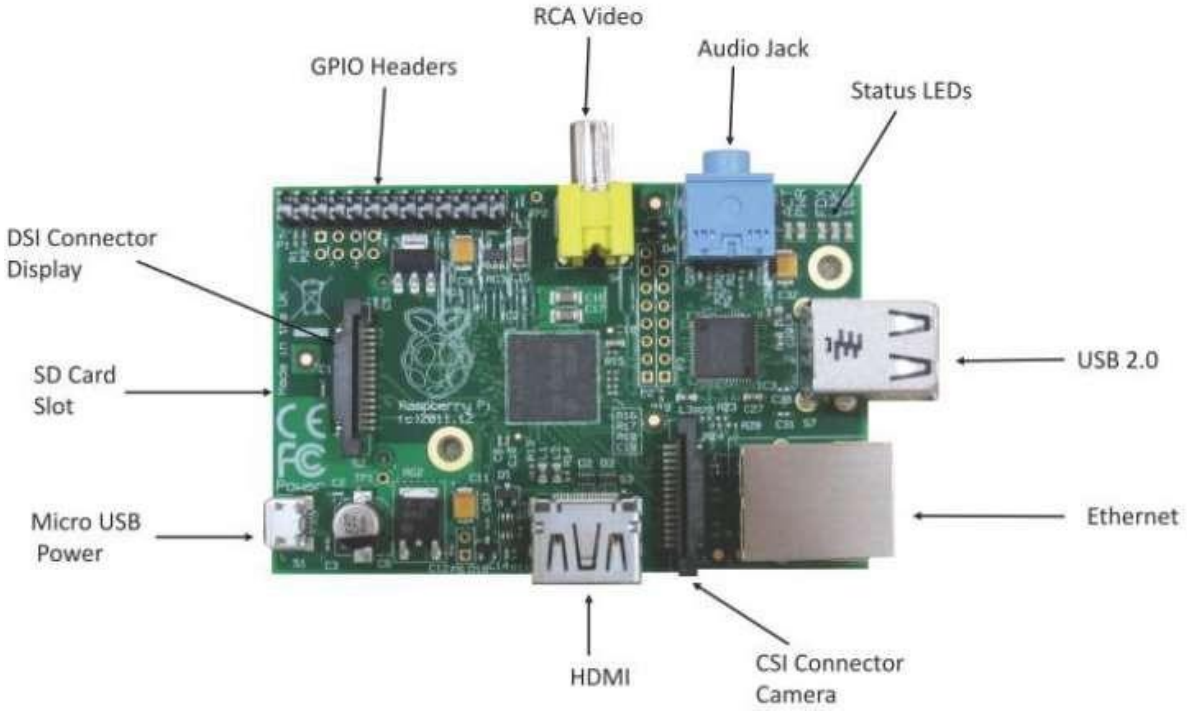
**Block diagram of an IoT Device:**




## Exemplary Device: Raspberry Pi:

1. Raspberry Pi is a low-cost mini-computer with the physical size of a credit card.
2. Raspberry Pi runs various flavors of Linux and can perform almost all tasks that a normal desktop computer can do.
3. Raspberry Pi also allows interfacing sensors and actuators through the general purpose I/O pins.
4. Since Raspberry Pi runs Linux operating system, it supports Python "out of the box".
5. Raspberry Pi is a low-cost mini-computer with the physical size of a credit card.
6. Raspberry Pi runs various flavors of Linux and can perform almost all tasks that a normal desktop computer can do.
7. Raspberry Pi also allows interfacing sensors and actuators through the general purpose I/O pins.
8. Since Raspberry Pi runs Linux operating system, it supports Python "out of the box".

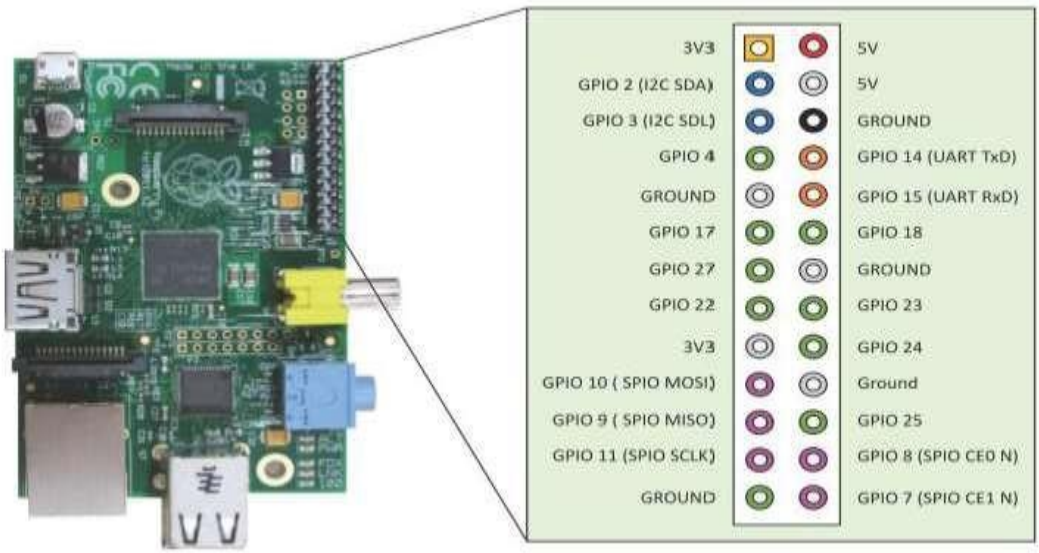
**Raspberry Pi:**




**Linux on Raspberry Pi:**

1. **Raspbian -** Raspbian Linux is a Debian Wheezy port optimized for Raspberry Pi.
2. **Arch -** Arch is an Arch Linux port for AMD devices.
3. **Pidora -** Pidora Linux is a Fedora Linux optimized for Raspberry Pi.
4. **RaspBMC -** RaspBMC is an XBMC media-center distribution for Raspberry Pi.
5. **OpenELEC -** OpenELEC is a fast and user-friendly XBMC media-center distribution.
6. **RISC OS -** RISC OS is a very fast and compact operating system.

**Raspberry Pi GPIO:**

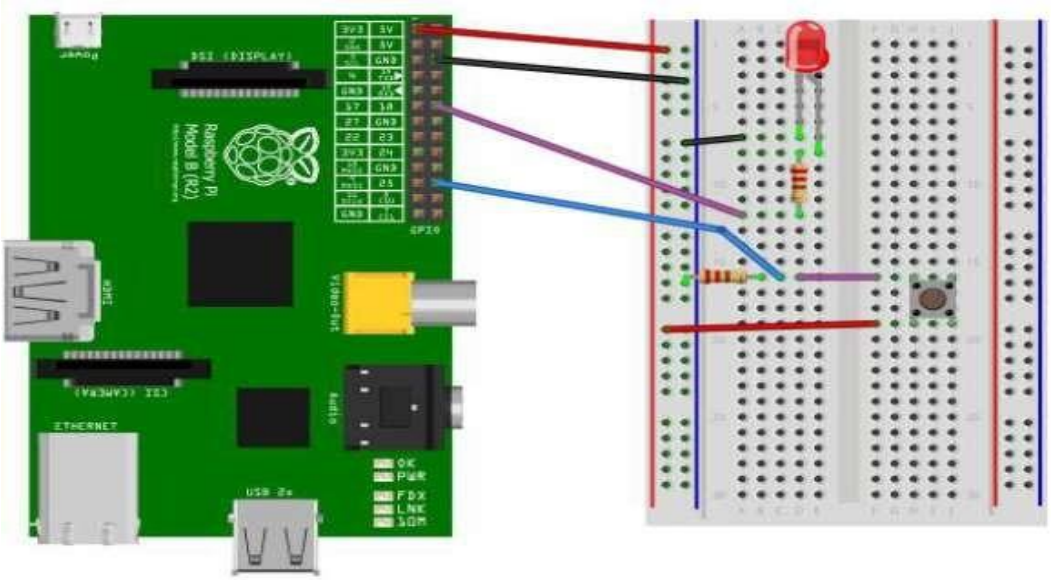


**Raspberry Pi Interfaces:**

1. **Serial -** The serial interface on Raspberry Pi has receive (Rx) and transmit (Tx) pins for communication with serial peripherals.
2. **SPI -** Serial Peripheral Interface (SPI) is a synchronous serial data protocol used for communicating with one or more peripheral devices.
3. **I2C -** The I2C interface pins on Raspberry Pi allow you to connect hardware modules. I2C interface allows synchronous data transfer with just two pins - SDA (data line) and SCL (clock line).

**Raspberry Pi Example: Interfacing LED and switch with Raspberry Pi:**

```
from time import sleep
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)

#Switch Pin
GPIO.setup(25, GPIO.IN)
#LED Pin
GPIO.setup(18, GPIO.OUT)
state=false

def toggleLED(pin):
        state = not state
        GPIO.output(pin, state)

while True:
        try:
                if (GPIO.input(25) == True):
                        toggleLED(pin)
                sleep(.01)
                except KeyboardInterrupt:
                        exit()
```

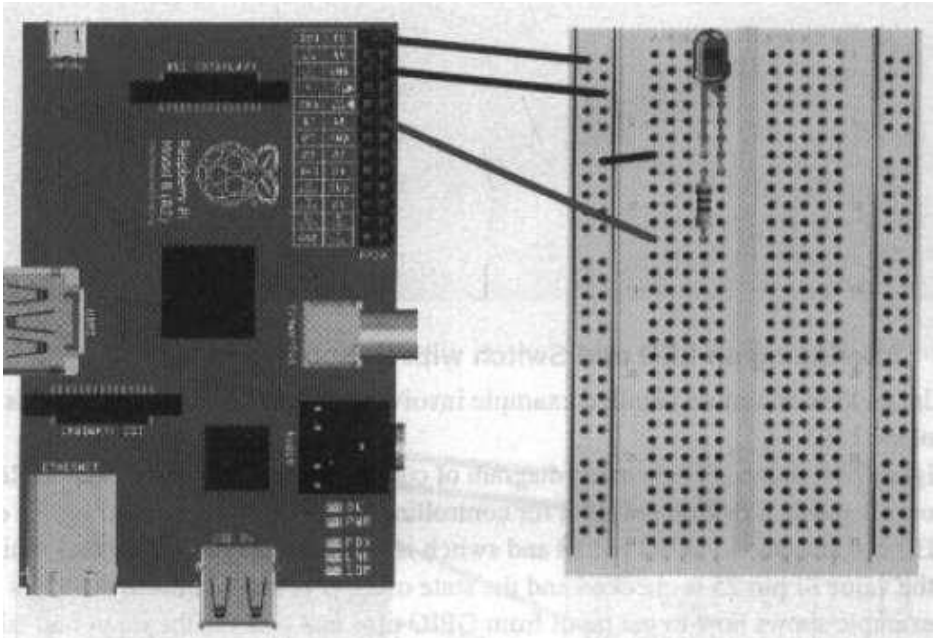


## Programming Raspberry Pi with Python:

Raspberry Pi runs on Linux and supports Python out of the box. We can run any Python program on a normal computer. The input/output capability provided by the GPIO pins on Raspberry Pi that makes it useful device for IoT. We can interface a wide variety of sensor and actuators with Raspberry Pi using the GPIO pins and the SPI, I2C and serial interfaces. Input from the sensors connected to Raspberry Pi can be processed and various actions can be taken, for instance, sending to data to a server, sending an email, triggering a relay switch.

**Controlling LED with Raspberry Pi:**

The following figure shows the schematic diagram of connecting an LED to Raspberry Pi.



The following code shows how to turn the LED on/off from command line. In this example LED is connected to GPIO pin 18. We cacn connect the LED to any other GPIO pin as well.

**Switching LED on/off from Raspberry Pi console**

```
$echo 18 > /sys/class/gpio/export
$cd /sys/class/gpio/gpio18

#Set pin 18 direction to out
$echo out > direction

#Turn LED on
$echo 1 > value
#Turn LED off
$echo 0 > value
```

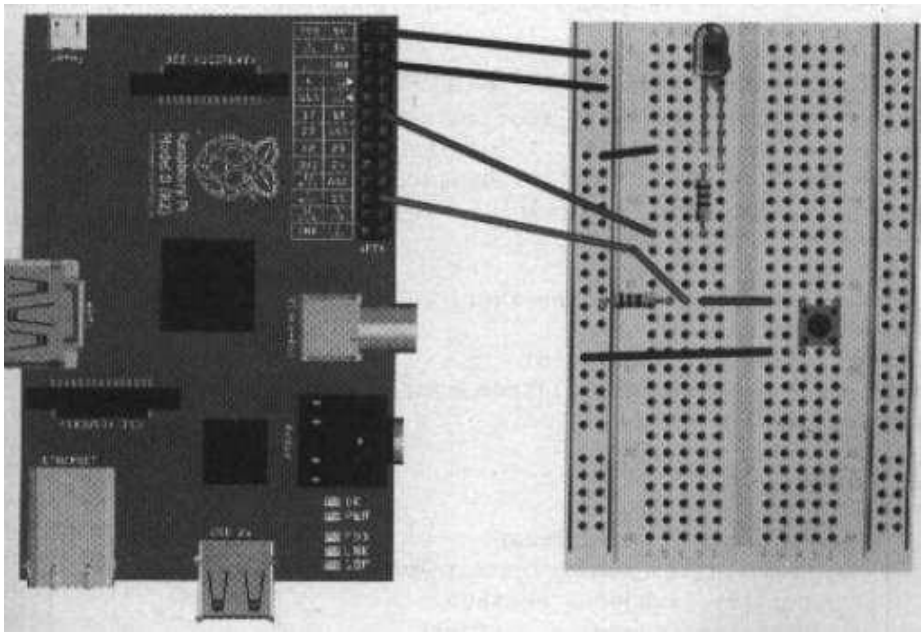
The following code shows a Python program for blinking an LED connected to Raspberry Pi every second. The program uses the RPi. GPIO module to control the GPIO ob Raspberry Pi. Inn this program, we set pin 18 direction to output and then write **True/False** alternatively after a delay of one second.

**Box 7.2: Python program for blinking LED**

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
GPIO.setup(18, GPIO.OUT)

while True:
   GPIO.output(18, True)
   time.sleep(1)
   GPIO.output(18, False)
   time.sleep(1)
```

**Interfacing an LED and Switch with Raspberry Pi:**

The following figure shows the schematic diagram of connecting an LED and switch to Raspberry Pi.



The following code shows a Python program for controlling an LED with a switch. In this example, the LED is connected to GPIO pin 18 and switch is connected to pin 25. In the infinite while loop the value of pin 25 is checked and the state of LED is is toggled if the switch is pressed. This example shows how to get input from GPIO pins and process the input and take some action. The action in this example is toggling the state of an LED.

**Python program for controlling an LED with a switch**

```
from time import sleep
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)
#Switch Pin
GPIO.setup(25, GPIO.IN)

#LED Pin
GPIO.setup(18, GPIO.OUT)

state=false

def toggleLED(pin):
   state = not state
   GPIO.output(pin, state)

while True:
   try:
      if (GPIO.input(25) == True):
         toggleLED(pin)
      sleep(.01)
   except KeyboardInterrupt:
      exit()
```

In another example, in which the action is an email alert. The following code shows a Python program for sending email on switch press. This program is similar to the above one. This program uses the Python SMTP library for sending an email when the switch connected to Raspberry Pi is pressed.

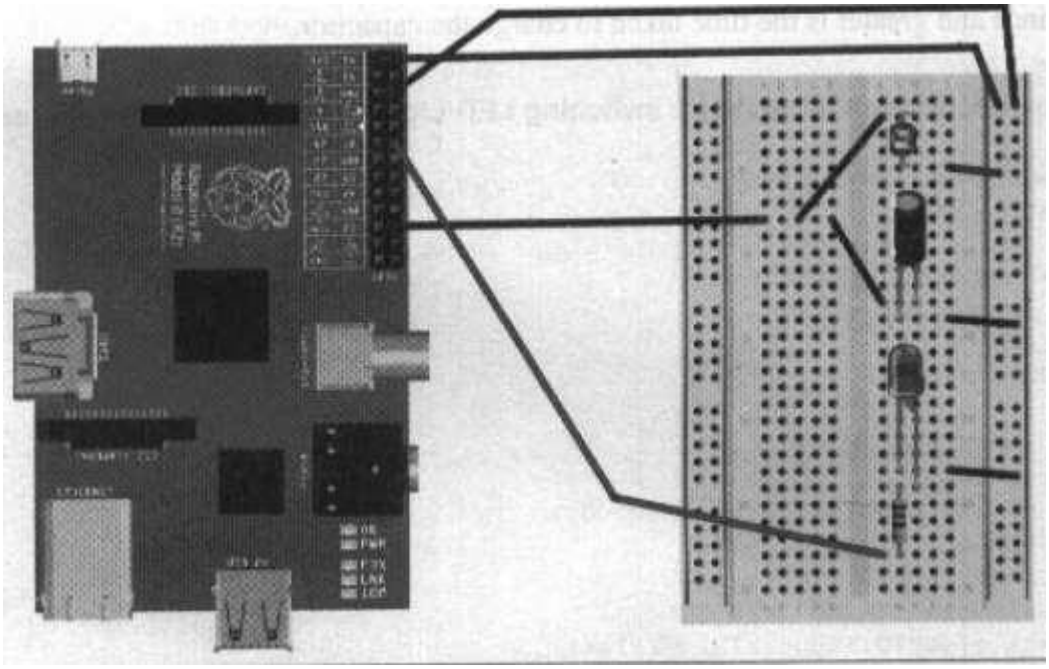
**Python program for sending an email on switch press**

```
import smtplib
from time import sleep
import RPi.GPIO as GPIO
from sys import exit

from_email = '<my-email>'
receipients_list = ['<receipient-email>']
cc_list = []
subject = 'Hello'
message = 'Switch pressed on Raspberry Pi'
username = '<Gmail-username>'
password = '<password>'
server = 'smtp.gmail.com:587'

GPIO.setmode(GPIO.BCM)
GPIO.setup(25, GPIO.IN)


def sendemail(from_addr, to_addr_list, cc_addr_list,
      subject, message,
      login, password,
      smtpserver):
```

```
    header = 'From:  %s \n' % from_addr
    header += 'To:  %s \n' % ','.join(to_addr_list)
    header += 'Cc:  %s \n' % ','.join(cc_addr_list)
    header += 'Subject:  %s \n \n' % subject
    message = header + message

    server = smtplib.SMTP(smtpserver)
    server.starttls()
    server.login(login,password)
    problems = server.sendmail(from_addr, to_addr_list, message)
    server.quit()

while True:
    try:
        if (GPIO.input(25) == True):
            sendemail(from_email, receipients_list,
              cc_list, subject, message,
              username, password, server)
        sleep(.01)
    except KeyboardInterrupt:
        exit()
```

**Interfacing a Light Sensor (LDR) with Raspberry Pi:**

Here we can discuss an example of interfacing a Light Dependent Resistor (LDR) with Raspberry Pi and turning on an LED on/off based on the light-level sensed. The following figure shows the schematic diagram of connecting an LDR to Raspberry Pi. Connect one side of LDR to 3.3V and other side to a 1µF capacitor and also to a GPIO pin (pin 18 in this example). AN LED is connected to pin 18 which is controlled based on the light-level sensed.



The following Python program is written for the LDR example. The **readLDR()** function returns a count which is proportional to the light level. In this function the LDR pin is set to output and low and then to input. At this point the capacitor starts charging through the resistor (and a counter is started) until the input pin reads high (this happens when capacitor voltage becomes greater than 1.4V). The counter is stopped when the input reads high. The final count is proportional to the light level as greater the amount of light, smaller is the LDR resistance and greater is the time taken to charge the capacitor.

**Python program for switching LED/Light based on reading LDR reading**

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
ldr_threshold = 1000
LDR_PIN = 18
LIGHT_PIN = 25

def readLDR(PIN):
   reading=0
   GPIO.setup(LIGHT_PIN, GPIO.OUT)
   GPIO.output(PIN, False)
   time.sleep(0.1)
   GPIO.setup(PIN, GPIO.IN)
   while (GPIO.input(PIN)==False):
      reading=reading+1
   return reading
def switchOnLight(PIN):
   GPIO.setup(PIN, GPIO.OUT)
   GPIO.output(PIN, True)

def switchOffLight(PIN):
   GPIO.setup(PIN, GPIO.OUT)
   GPIO.output(PIN, False)


while True:
   ldr_reading = readLDR(LDR_PIN)
   if ldr_reading < ldr_threshold:
      switchOnLight(LIGHT_PIN)
   else:
      switchOffLight(LIGHT_PIN)

   time.sleep(1)
```

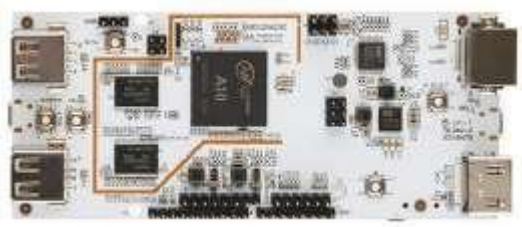
## Other Devices:

**1. pcDuino:**

pcDuino is an Arduino-pin compatible single board mini-computer that comes with a 1 GHz ARM Cortex-A8 processor. pcDuino is a high performance and cost effective device that runs PC like OS such as Ubuntu and Android ICS. Like. Raspberry Pi, it has an HDMI video/audio interface. pcDuino supports various programming languages including C, C++ (with GNU tool chain), Java (with standard Android SDK) and Python.

**2. BeagleBone Black: 189**

BeagleBone Black is similar to Raspberry Pi, but a more powerful device. It comes with a 1 GHZ ARM Corex-A8 processor and supports both Linux and Android operating systems. Like Raspberry Pi, it has HDMI video/audio interface, USB and Ethernet ports.



**3. Cubieboard:**

Cubieboard is powered by dual core ARM Cortex A7 processor and has a range of input/output interfaces including USB, HDMI, IR, serial, Ethernet, SATA, and a 96 pin extended interface. Cubieboard also provides SATA support. The board can run both Linux and Android operating systems.

# IOT PHYSICAL SERVERS & CLOUD OFFERINGS

## Introduction to Cloud Storage Models & Communication APIs:

Cloud Computing is a transformative computing paradigm that involves delivering applications and services over the internet. National Institute of Standards & Technology (NIST) defines cloud computing as:

"Cloud Computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider transaction."

## WAMP - AutoBahn for IoT:

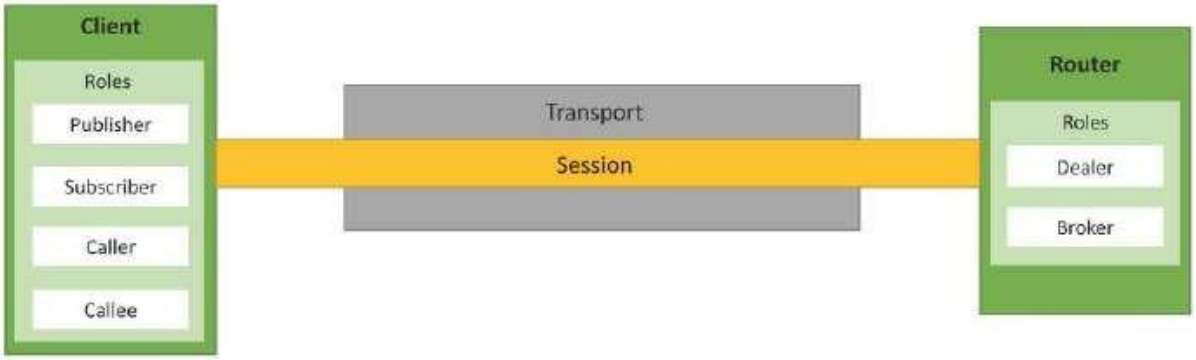
Web Application Messaging Protocol (WAMP) is a sub-protocol of WebSocket which provides publish-subscribe and remote procedure call (RPC) messaging patterns. WAMP enables distributed application architectures where the application components are distributed on multiple nodes and communicate with messaging patterns provided by WAMP. The key concepts of WAMP are:

1. **Transport:** Transport is channel that connects two peers.
2. **Session:** Session is a conversation between two peers that runs over a transport.
3. **Client:** Clients are peers that can have one or more roles. In publish-subscribe model client can have following roles:
   - **Publisher:** Publisher publishes events (including payload) to the topic maintained by the Broker.
   - **Subscriber:** Subscriber subscribes to the topics and receives the events including the payload.
4. In RPC model client can have following roles:
   - **Caller:** Caller issues calls to the remote procedures along with call arguments.
   - **Callee:** Callee executes the procedures to which the calls are issued by the caller and returns the results back to the caller.
5. **Router:** Routers are peers that perform generic call and event routing. In publish-subscribe model Router has the role of a Broker:
   - **Broker:** Broker acts as a router and routes messages published to a topic to all subscribers subscribed to the topic.
6. In RPC model Router has the role of a Broker:

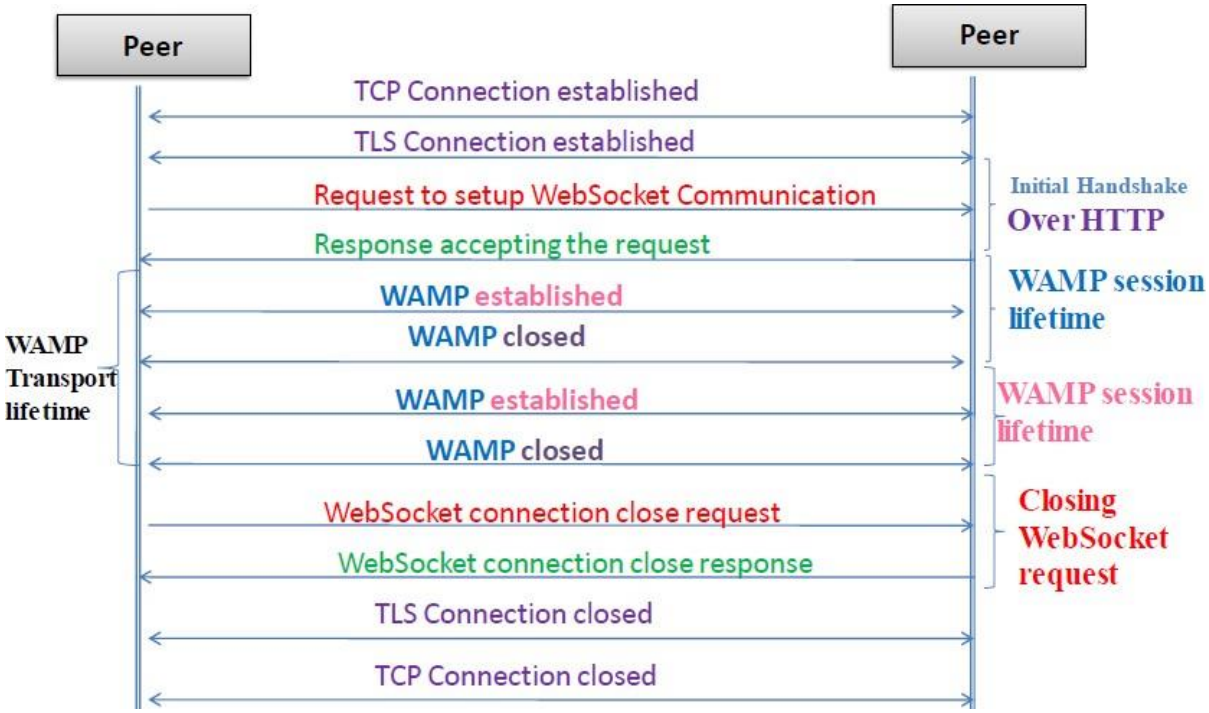
– **Dealer:** Dealer acts a router and routes RPC calls from the Caller to the Callee and routes results from Callee to Caller.

7. **Application Code:** Application code runs on the Clients (Publisher, Subscriber, Callee or Caller).

The following figure shows a WAMP Session between Client and Router, established over a Transport.

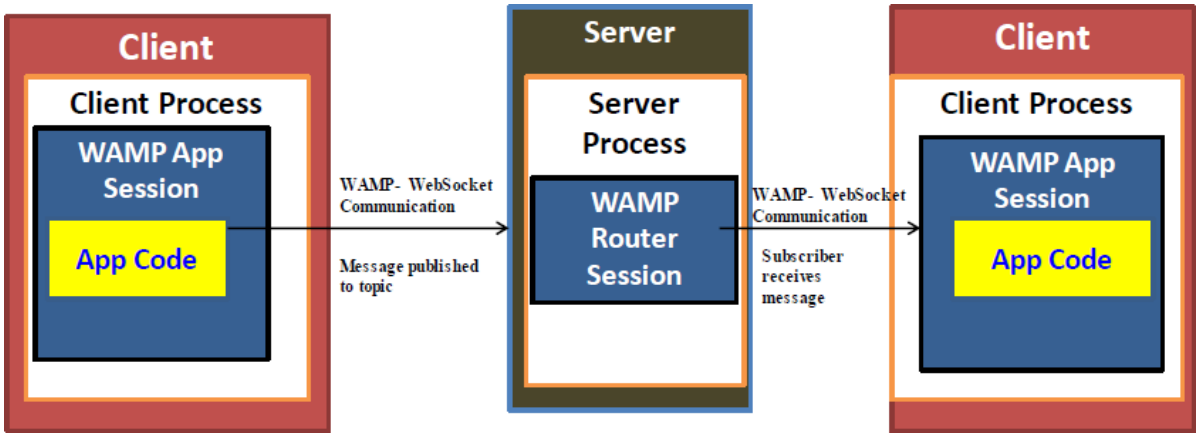



The following figure shows the WAMP protocol interactions between peers. The WAMP transport used is WebSocket. WAMP sessions are established over WebSocket transport within the lifetime of WebSocket transport.




The following figure shows the communication between various components of a typical WAMP-AutoBahn deployment. The client (in Publisher role) runs a WAMP application component that publishes messages to the Router. The Router (in Broker role)

decouples the publisher from the subscribers. The communication between Publisher – Broker and Broker – Subscribers happens over a WAMP-WebSocket session.




An example of a WAMP publisher and subscriber implemented using AutoBahn. The following steps explains to work with AutoBahn.

**Step 1:** Commands for Installing AutoBahn

```
sudo apt-get install python-twisted python-dev
sudo apt-get install python-pip
sudo pip install -upgrade twisted
sudo pip install -upgrade autobahn
```

**Step 2:** After installing AutoBahn, Clone AutoBahn Python from github

```
git clone https://github.com/tavedo/AutobahnPython.git
```

**Step 3:** Create a WAMP publisher component as the following. The publisher component publishes a message containing the current timestamp to a topic named "test-topic"

**Example of a WAMP Publisher implemented using AutoBahn framework - publisherApp.py**

```
from twisted.internet import reactor
from twisted.internet.defer import inlineCallbacks
from autobahn.twisted.util import sleep
from autobahn.twisted.wamp import ApplicationSession
import time,datetime
def getData():
   #Generate message
   timestamp = datetime.datetime.fromtimestamp(
   time.time()).strftime('%Y-%m-%d%H:%M:%S')
   data = "Message at time-stamp:  "+str(timestamp)
   return data

#An application component that publishes an event every second.
class Component(ApplicationSession):
   @inlineCallbacks
   def onJoin(self, details):
      while True:
         data = getData()
         self.publish('test-topic', data)
         yield sleep(1)
```

**Step 4:** Create a WAMP subscriber component as the following. The subscriber component that subscribes to the "test-topic"

**Example of a WAMP Subscriber implemented using AutoBahn framework - subscriberApp.py**

```
from twisted.internet import reactor
from twisted.internet.defer import inlineCallbacks
from autobahn.twisted.wamp import ApplicationSession

#An application component that subscribes and receives events
class Component(ApplicationSession):
   @inlineCallbacks
   def onJoin(self, details):
      self.received = 0

      def on_event(data):
         print "Received message:   " + data
         yield self.subscribe(on_event, 'test-topic')

   def onDisconnect(self):
      reactor.stop()
```

**Step 5:** Run the application router on a WebSocket transport server as follows.

```
▪ python AutobahnPython/examples/twisted/wamp/basic/server.py
```

Run the publisher component over a WebSocket transport client as follows:

```
▪ python AutobahnPython/examples/twisted/wamp/basic/client.py –component
"publisherApp.Component"
```

Run the subscriber component over a WebSocket transport client as follows:

```
▪ python AutobahnPython/examples/twisted/wamp/basic/client.py –component
"subscriberApp.Component"
```

While we can setup the server and client processes on a local machine for trying out the publish-subscribe example, in production environment, these components run on separate machines. The server process (the brains or "Thing Tank"!) is setup on a cloud-based instance while the client processes can run either on local hosts/devices or in the cloud.

## Xively Cloud for IOT:

### What is Xively?

1. Xively is a commercial Platform-as-a-Service
2. Xively can be used for creating solutions for Internet of Things
3. With Xively cloud, IoT developers can focus on
    a. the front end infrastructure
    b. Devices for IoT
    c. Management of back end data collection infrastructure

**Xively Platform:** This consists of

1. Message bus for real time message management
2. Data Services for time series archiving
3. Directory services for device provisioning and management

**Advantages of Xively:**

1. Xively provides an extensive support for various languages and platforms
2. Xively libraries leverage standards based API over HTTP, Sockets and MQTT for connecting IoT devices to the Xively Cloud

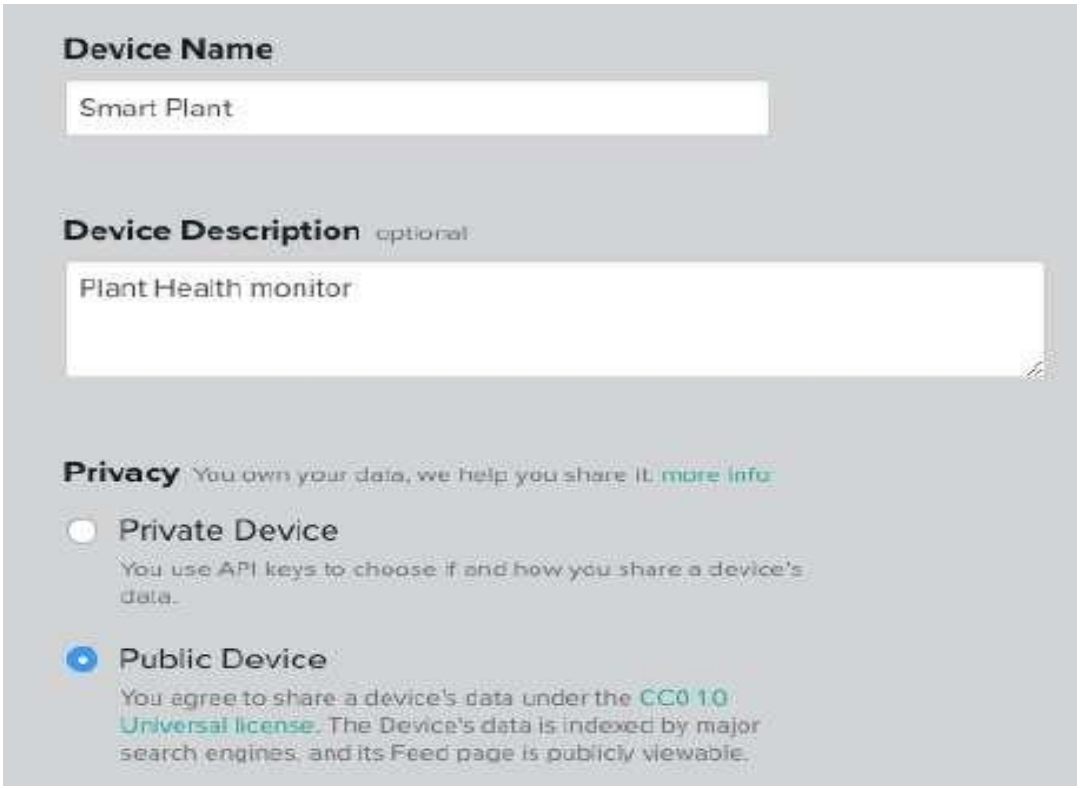
**How to use Xively?**

1. To start using Xively, one need to register for a developer account
2. Then, create development device on Xively
3. When device is created, Xively automatically creates **a <u>Feed-ID and an API Key</u>** to connect to the device
    a. Each device has s unique **<u>Feed-ID</u>**.

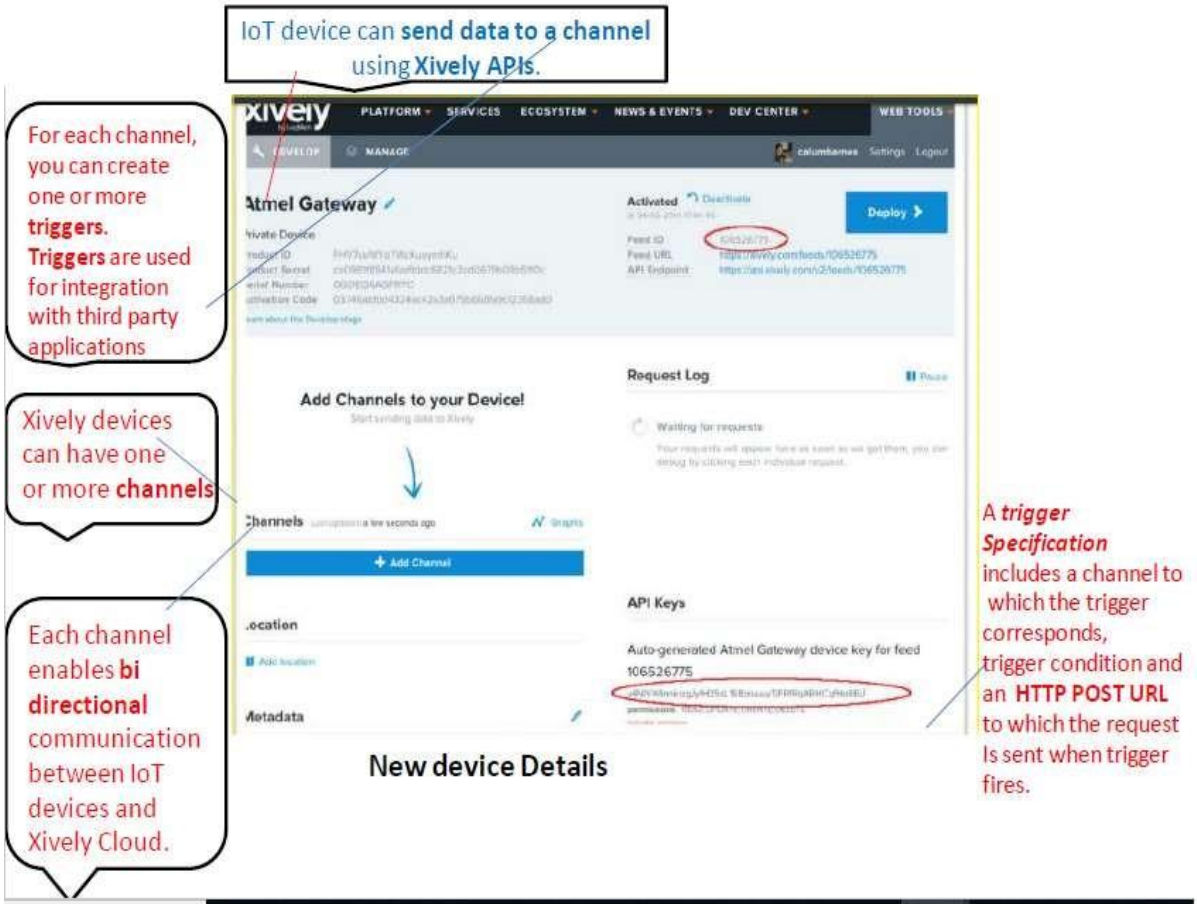
"Feed-ID is a **collection of channels** or datastreams defined for a device and the associates meta-data"

"API keys are used to provide different levels of **permissions**. The default API key has **read, update, create** and **delete** permissions"

**Xively Cloud:** Dashboard: The following figure shows to create a new device (Add Device).



The following figure shows new device details.



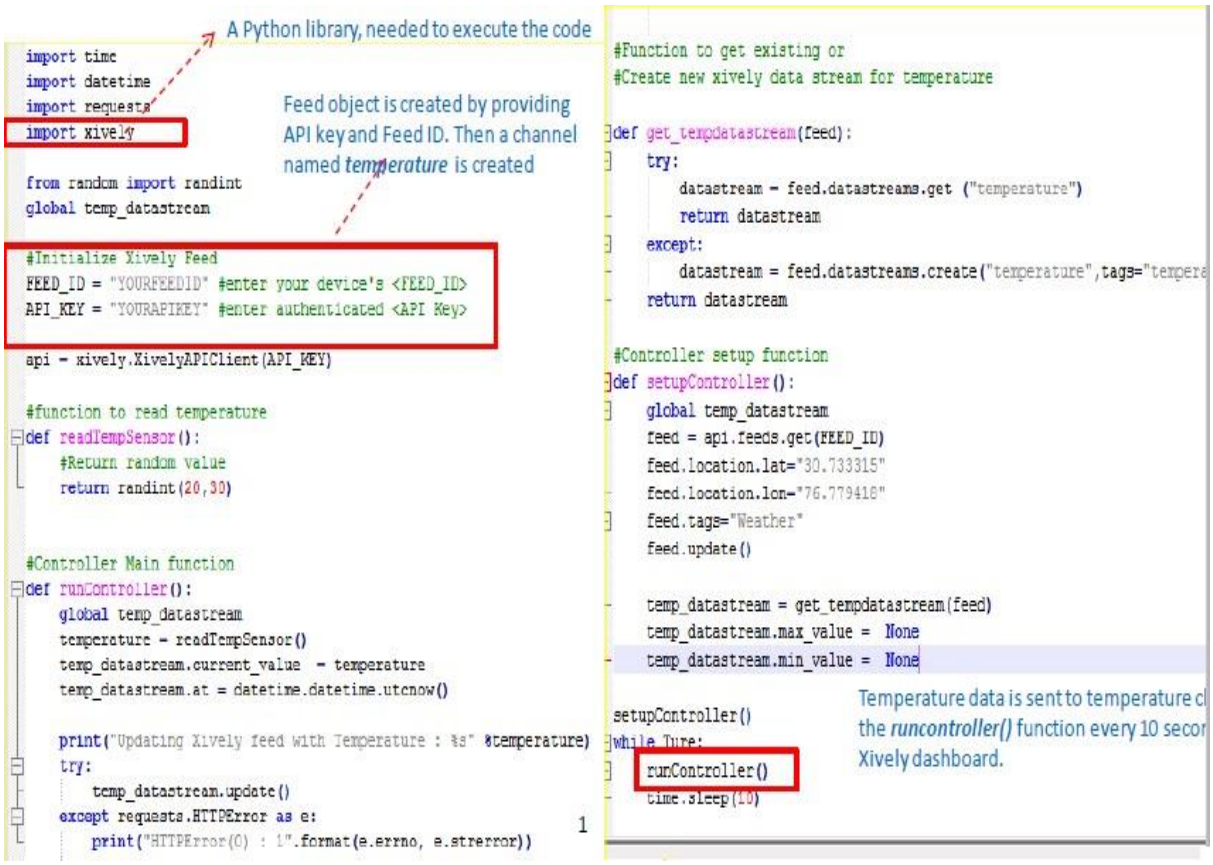

New device Details

**Xively Cloud: Python Program for sending data to a Xively Cloud:**

The following **figure (a)** shows the Python program for the sending temperature data to Xively Cloud. This example uses the Xively Python Library.

**Background:** Temperature monitoring using Raspberry Pi and the measured data is sent to Xively cloud. Raspberry Pi runs a controller program that reads the sensor data (e.g.DHT11) every few seconds and sends data to the cloud.

The temperature data is sent the channel in the **runController()** function for every 10 seconds. The following **figure (b)** shows the temperature channel in the Xively dashboard. In this example, a single Xively device with one channel is created. In real-world scenario each Xively device can have multiple channels and we can have multiple devices in a production batch.




```
import time
import datetime
import requests
import xively

from random import randint
global temp_datastream

#Initialize Xively Feed
FEED_ID = "YOURFEEDID" #enter your device's <FEED_ID>
API_KEY = "YOURAPIKEY" #enter authenticated <API Key>

api = xively.XivelyAPIClient(API_KEY)

#function to read temperature
def readTempSensor():
    #Return random value
    return randint(20,30)


#Controller Main function
def runController():
    global temp_datastream
    temperature = readTempSensor()
    temp_datastream.current_value  = temperature
    temp_datastream.at = datetime.datetime.utcnow()

    print("Updating Xively feed with Temperature : %s" %temperature)
    try:
        temp_datastream.update()
    except requests.HTTPError as e:
        print("HTTPError(0) : 1".format(e.errno, e.strerror))
```

```
#Function to get existing or
#Create new xively data stream for temperature

def get_tempdatastream(feed):
    try:
        datastream = feed.datastreams.get ("temperature")
        return datastream
    except:
        datastream = feed.datastreams.create("temperature",tags="tempera
    return datastream

#Controller setup function
def setupController():
    global temp_datastream
    feed = api.feeds.get(FEED_ID)
    feed.location.lat="30.733315"
    feed.location.lon="76.779418"
    feed.tags="Weather"
    feed.update()

    temp_datastream = get_tempdatastream(feed)
    temp_datastream.max_value = None
    temp_datastream.min_value = None

setupController()
while True:
    runController()
    time.sleep(10)
```

**Figure (a)**

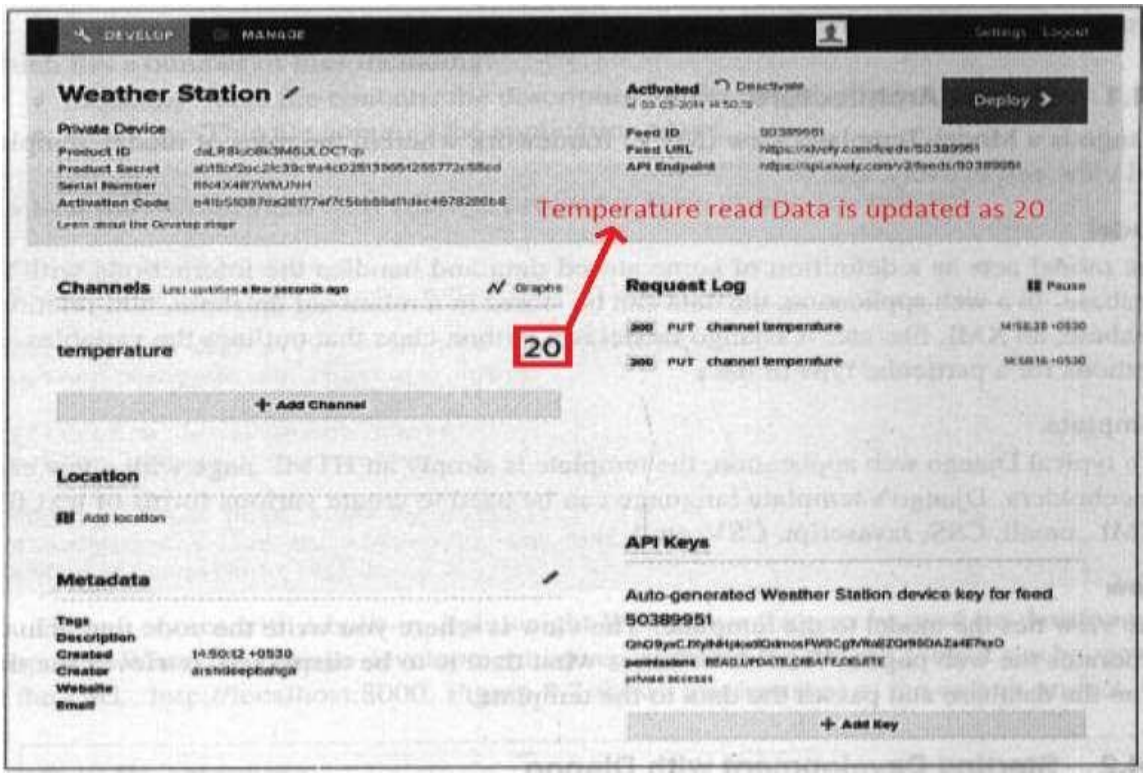



**Figure (b)**

## Python Web Application Framework – Django:

### What is Django?

Django is an open source web application framework for developing web applications in Python.

### What is web application framework?

Web application framework is a collection of solutions, packages and best practices that allows development of web applications and dynamic websites.

### Django Architecture:

1. Django is based on Model-Template-View architecture
2. Django provides separation of the data model from business rules and the user interface
3. Django provides a unified API to a database backend
4. Web application built with Django can work with different databases without requiring any code changes
5. With the web application design combined with powerful capabilities of Python language and Python ecosystem => Django best suited for IoT applications.
6. Django consists of an object-relational mapper, a web templating system and regular expression based URL dispatcher

Django is based on **Model-Template-View** architecture and is explained as below:

| Model | Template | View |
|---|---|---|
| 1. Acts as a definition of some stored data.<br>2. It handles interactions with the database.<br>3. In Web Application, the data can be stored in a relational database, non-relational database, an XML file etc.<br>4. A Django model is a Python class that outlines the variables and methods for a particular type of data. | 1. The Template is simply an HTML page with a few extra placeholders.<br>2. Django's template language can be used to create various forms of text files (XML, email, CSS, JavaScript, and CSV etc.). | 1. The View ties the model to the template.<br>2. The view is where we write the code that actually generates the web pages.<br>3. View determines what data is to be displayed, retrieves the data from the database and passes the data to the template. |

**Starting Development with Django:**

The following links provide the instructions for setting up Django. We will learn how to start developing web applications with Django.

**Preferred:**

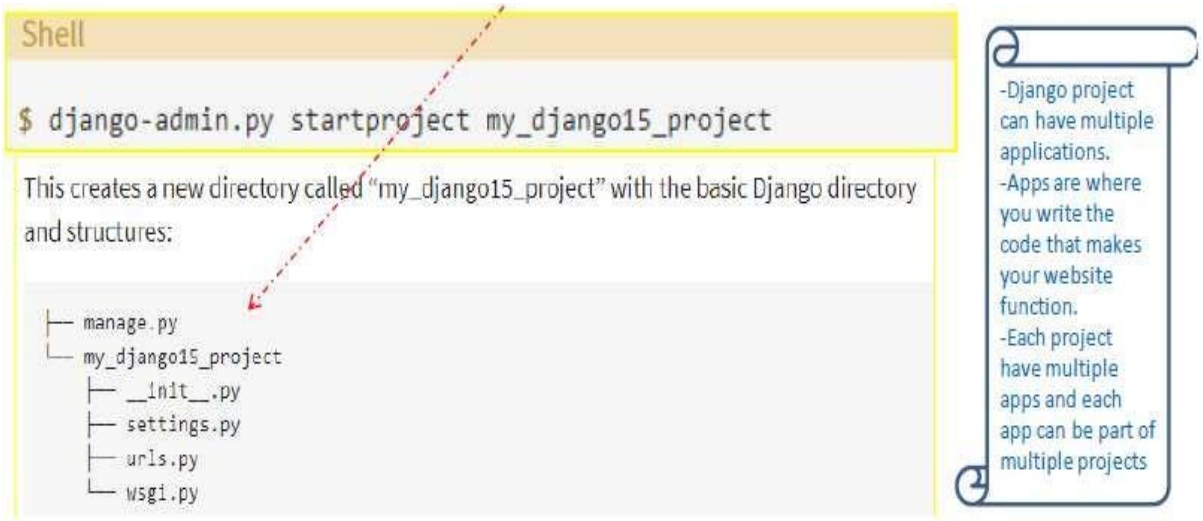
1. https://realpython.com/django-setup/
2. https://docs.djangoproject.com/en/3.0/topics/install/
3. https://medium.com/analytics-vidhya/django-rest-api-with-json-web-token-jwt-authentication-69536c01ee18

**Others:**

1. https://github.com/Huachao/azure-content/blob/master/articles/virtual-machines/virtual-machines-python-django-web-app-linux.md
2. https://azure.microsoft.com/en-in/blog/using-django-python-and-mysql-on-windows-azure-web-sites-creating-a-blog-application/
3. https://azure.microsoft.com/en-in/resources/templates/django-app/
4. https://docs.djangoproject.com/en/3.0/intro/tutorial01/

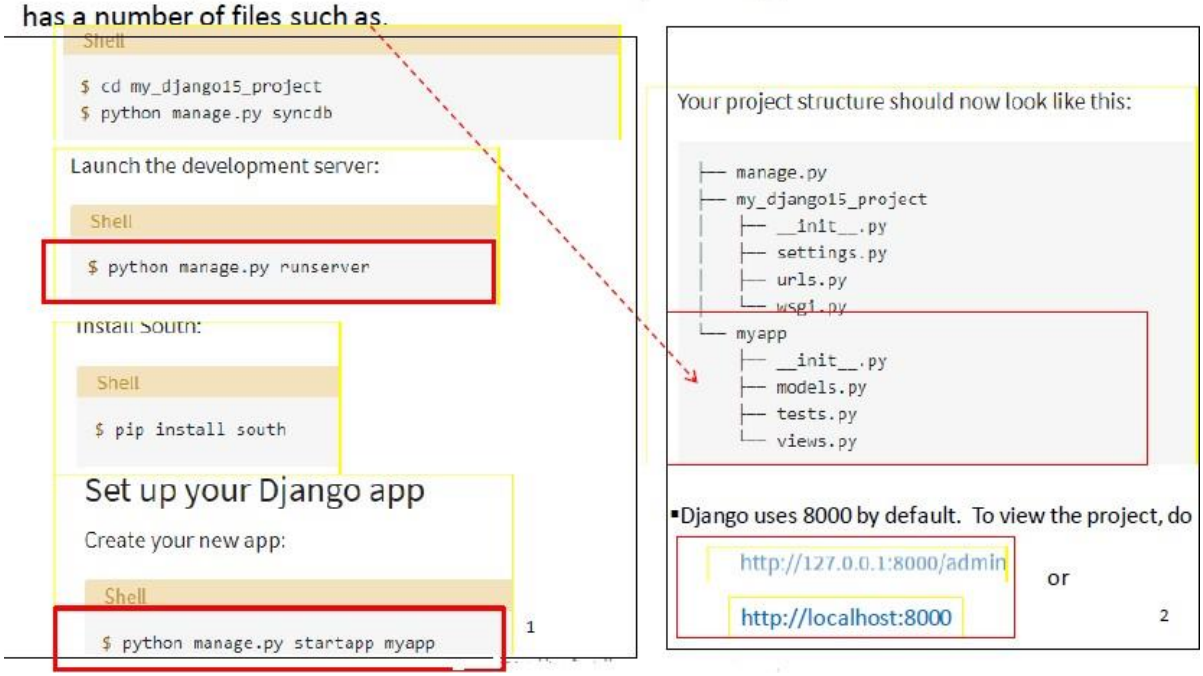
**Creating a Django Project and App:**

Step 1: When you create a new Django project, a number of files are created as shown below:




Step 2: (next page)

When a new application is created a new directory for the application is also created which has a number of files such as.




Step 3: **Configuring a Database**:

a. Most web applications have a database backend.

b. Developers have a wide choice of databases that can be used for web applications including both relational and non-relational databases.

c. Django provides a unified API for database backbends thus giving the freedom to choose the database.

d. Django supports various relational database engines including MySQL, PostgreSQL, Oracle and SQLite3.

e. Django supports non-relational databases such as MongoDB. MongoDBcan be added by installing additional engines.

Step 4: **Setting up MySQL database**:

```
sudo apt-get install mysql-server mysql-client
sudo mysqladmin -u root -h localhost password 'password'
```

example

```
(base) C:\Users\CEDLAB>pip3 install mysqlclient
Collecting mysqlclient
  Downloading mysqlclient-1.4.6-cp38-cp38-win_amd64.whl (263 kB)
     |████████████████████████████████| 263 kB 656 kB/s
Installing collected packages: mysqlclient
Successfully installed mysqlclient-1.4.6
```

Step 5: **Configuring MySQL database with Django – settings.py**:

@DATABASE dictionary

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'DB_NAME',
        'USER': 'DB_USER',
        'PASSWORD': 'DB_PASSWORD',
        'HOST': 'localhost',   # Or an IP Address that your DB is hosted on
        'PORT': '3306',
    }
}
```

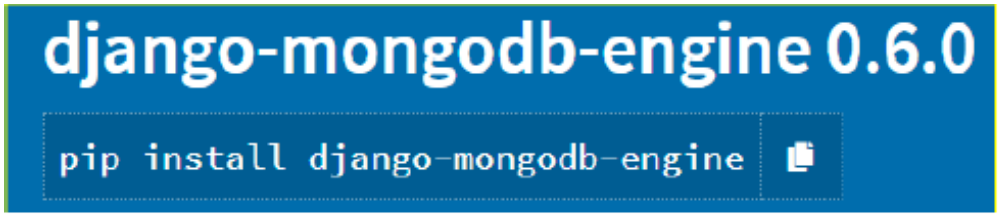
You also have the option of utilizing MySQL option files, as of Django 1.7. You can accomplish this by setting your `DATABASES` array like so:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'OPTIONS': {
            'read_default_file': '/path/to/my.cnf',
        },
    }
}
```

You also need to create the `/path/to/my.cnf` file with similar settings from above

```
[client]
database = DB_NAME
host = localhost
user = DB_USER
password = DB_PASSWORD
default-character-set = utf8
```

Step 6: **Setting up MongoDB and Django-MongoDB engine**:




Django-nonrel

```
pip install git+https://github.com/django-nonrel/django@nonrel-1.5
```

djangotoolbox

```
pip install git+https://github.com/django-nonrel/djangotoolbox
```

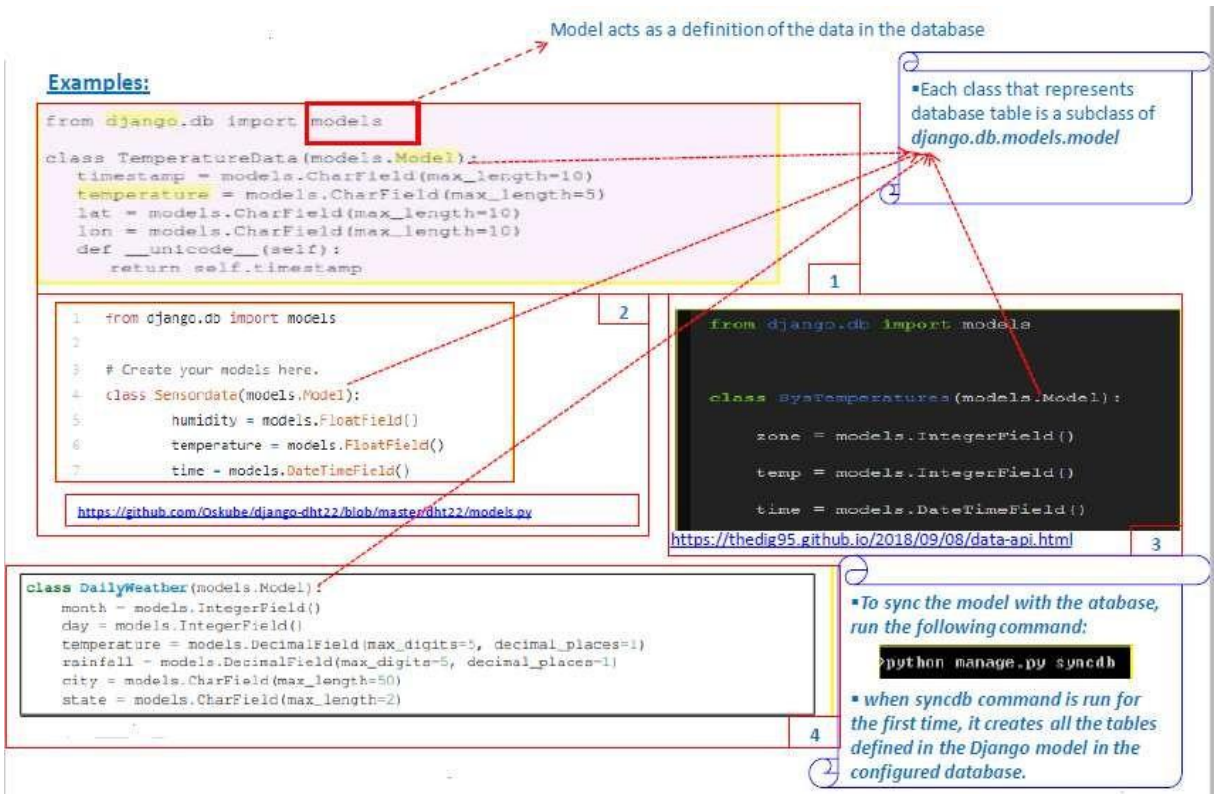
Django MongoDB Engine

You should use the latest Git revision.

```
pip install git+https://github.com/django-nonrel/mongodb-engine
```

Step 7: **Configuring MongoDB with Django- settings.py**: (next page)


Step 8: **Defining a Model**:




Step 9: **Django Admin site**:

a. Django provides an administration system that allows managing the website without writing additional code.

b. "admin" system reads the Django model and provides an interface that can be used to add content to site

c. The Django admin site is enabled by adding django.contrib.admin and django.contrib.admindocs to the INSTALLED_APPS section in the settings.py file

d. The admin site requires URL pattern definitions in the urls.py

e. To define the editable application models in the admin interface, a new file named **admin.py** is to be created.

**Enabling admin for Django Models**

```
from django.contrib import admin
from myapp.models import TemperatureData

admin.site.register(TemperatureData)
```

Step 10: **Adding new items in the temperature data table using admin site**:


Step 11: **Adding new items in the data table using admin site**:


Step 12: **Defining a View**:

a. The view contains the logic that glues the model to the template.

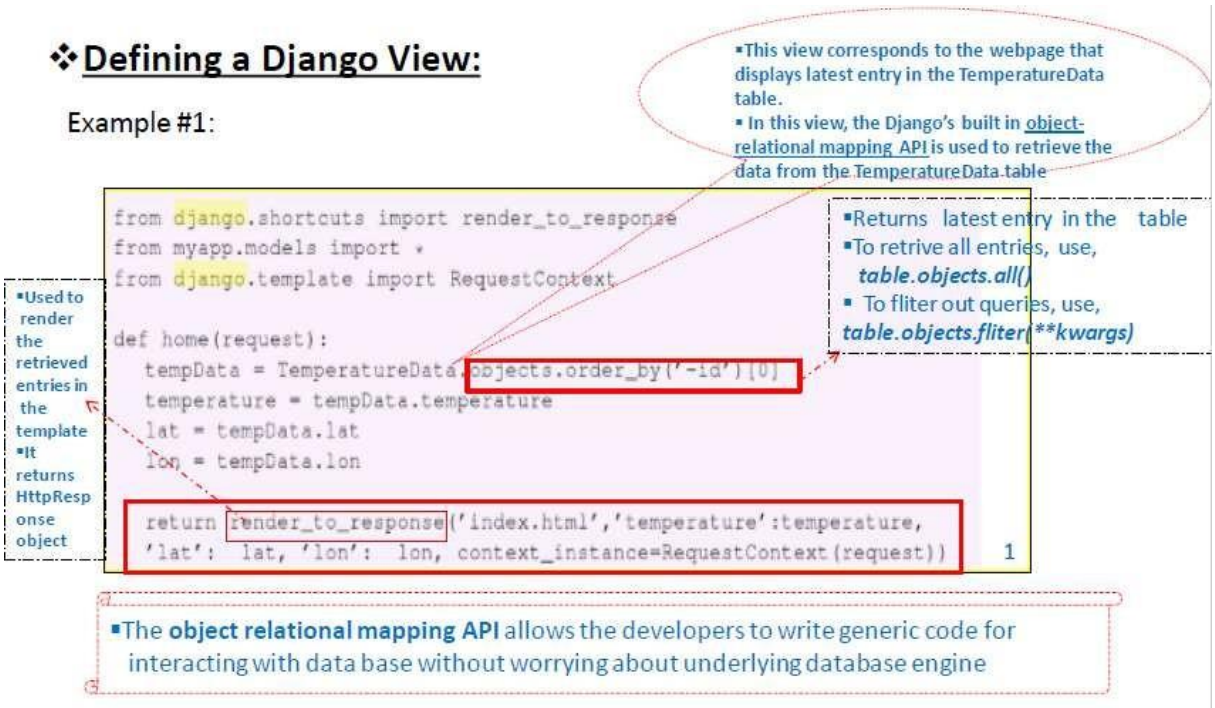
b. The view determines the data to be displayed in the template, retrieves the data from the Database and passes it to the template

c. The view also extracts the data posted in a form in the template and inserts it in the database

d. Each page in the website has a separate view, and is basically a Python function in the views.py file

e. Views can also perform additional tasks such as authentication, sending emails etc.

Step 13: **Defining a Django View**:




## ❖Defining a Django View:

Example #2:

```
from django.shortcuts import render
from django.http import HttpResponse, JsonResponse, Http404
from django.template import Template, Context
from django.utils import timezone
from django.core import serializers
from dht22.models import Sensordata

# Create your views here.
def index(request):
    return render(request, "dht22/chart.html")

# Sends latest humidity and temperature values
def currentview(request):
    sensordata = Sensordata.objects.order_by('-time')[0]
    context = {
        'humidity': sensordata.humidity,
        'temperature': sensordata.temperature,
    }

    return JsonResponse({'humidity': sensordata.humidity, 'temperature': sensordata.temperature})
```

2

## ❖ Defining a Django View:

Example #3:

```
import requests
from django.shortcuts import render
from .models import City

def index(request):
    url = 'http://api.openweathermap.org/data/2.5/weather?q=
{}&units=imperial&appid=271d1234d3f497eed5b1d80a07b3fcd1'
    city = 'Las Vegas'

    cities = City.objects.all()

    weather_data = []

    for city in cities:
        r = requests.get(url.format(city)).json()

        city_weather = {
            'city': city.name,
            'temperature': r["main"]["temp"],
            'description': r["weather"][0]["description"],
            'icon': r["weather"][0]["icon"],
        }
        weather_data.append(city_weather)

    context = {'weather_data' : weather_data}
    return render(request, 'weather/weather.html', context)
```

3

## ❖ Defining a Django View:

Example #4:

the_weather/weather/views.py

```
from django.shortcuts import render
import requests

def index(request):
    url = 'http://api.openweathermap.org/data/2.5/weather?q={}&units=imperial&appid=YOUR_APP_KEY'
    city = 'Las Vegas'
    city_weather = requests.get(url.format(city)).json() #request the API data and convert the JS

    return render(request, 'weather/index.html') #returns the index.html template
```

4

## ❖Defining an alternative Django View that retrieves data from Xively cloud

```
from django.shortcuts import render_to_response
from django.template import RequestContext
import requests
import xively

FEED_ID = "<enter-id>"
API_KEY = "<enter-key>"
api = xively.XivelyAPIClient(API_KEY)

feed = api.feeds.get(FEED_ID)
temp_datastream = feed.datastreams.get("temperature")
```

```
def home(request):
    temperature=temp_datastream.current_value
    lat=feed.location.lat
    lon=feed.location.lon

    return render_to_response('index.html','temperature':temperature,
    'lat': lat, 'lon': lon, context_instance=RequestContext(request))
```

Step 14: **Defining a Django Template**:

a. A Django template is typically an HTML file

b. Django template allow separation of the presentation of data from the actual data using placeholder and associated logic(using template tags)

c. A template receives a context from the view and presents the data in context variables in the placeholders.

**Example #1:**

the_weather/weather/templates/weather/index.html

```
<div class="box">
    <article class="media">
        <div class="media-left">
            <figure class="image is-50x50">
                <img src="http://openweathermap.org/img/w/{{ weather.icon }}.png" alt="Image">
            </figure>
        </div>
        <div class="media-content">
            <div class="content">
                <p>
                    <span class="title">{{ weather.city }}</span>
                    <br>
                    <span class="subtitle">{{ weather.temperature }}° F</span>
                    <br> {{ weather.description }}
                </p>
            </div>
        </div>
    </article>
</div>
```

**Example #2- A template for the weather station app**

```
<html>
<head>
<meta charset="utf-8">
<link href="/static/css/bootstrap-responsive.css" rel="stylesheet">
<script type="text/javascript"
src="http://maps.google.com/maps/api/js?sensor=false"></script>
<script type="text/javascript">
function initialize()
   var latlng = new google.maps.LatLng(lat,lon);
   var settings =
   zoom: 11,
   center: latlng,
   mapTypeControl: false,
   mapTypeControlOptions: style:
google.maps.MapTypeControlStyle.DROPDOWN_MENU,
   navigationControl: true,
   navigationControlOptions: style:
google.maps.NavigationControlStyle.SMALL,
   mapTypeId: google.maps.MapTypeId.TERRAIN
   ;

var map = new google.maps.Map(document.getElementById("map_canvas"),
settings);
var vespiMarker = new google.maps.Marker(
   position: latlng,
   map: map,
```

1

```
   title:"CityName, "
   );
   startws ();

</script>
<title>Weather Station</title>
</head>
<body onload="initialize()">

<div class="container">
   <center><h1>Weather Station</h1></center>
   <h3>CityName</h3>
   <div class='row'>
   <div class='span3'><h4>Temperature</h4></div>
   <div class='span3'>
<h4 id='temperature'>{{temperature}}</h4></div></div>
   <div class="span6" style="height:435px">
   <div id = 'map_canvas">
   </div>
   </div>
</div>

</body>
</html>
```

2

Step 15: **Defining the Django URL patterns**:

a. URL patterns are a way of mapping the URLs to the views that should handle the URL requests

b. URLs requested by the user are matched with the URL patterns

c. The view corresponding to the pattern matches the URL is used to handle the request.

**Example of URL configuration:**

```
from django.conf.urls.defaults import *
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
   url(r'^$', 'myapp.views.home', name='home'),
   url(r'^admin/doc/', include('django.contrib.admindocs.urls')),
   url(r'^admin/', include(admin.site.urls)),
)
```

1

## Example of URL configuration:

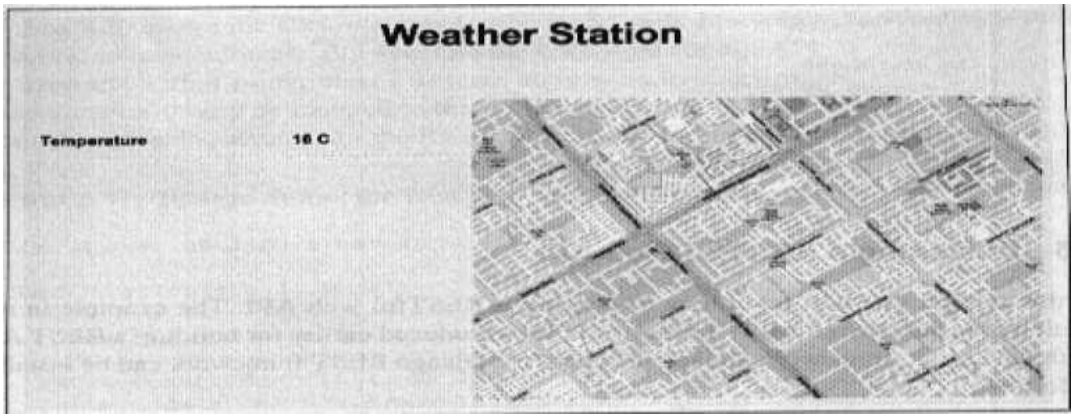
```
"""weatherapp URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/2.2/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.urls import include, path
    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
"""
from django.contrib import admin
from django.urls import path,include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('',include('weather.urls')),
]
```

2

Step 16: With the Django Models, Views, Templates and URLs defined for the Django project, the application will finally run with commands as shown below:

```
#cd into the project root
$cd weatherStationProject

#Sync the database
$ python manage.py syncdb

#Run the application
$ python manage.py runserver
```

Step 17: The weather station app Screen shot:



## Designing a RESTful Web API:

1. With the Django framework already installed, the Django REST framework can be installed as follows:

```
pip install djangorestframework
pip install markdown
pip install django-filter
```

2. After installing the Django REST framework, let us create a new Django project named ***restfulapi***, and then start a new app called ***myapp*** as follows:

```
django-admin.py startproject restfulapi
cd restfulapi
```

```
python manage.py startapp myapp
```

3. REST API allows creating, viewing, updating and deleting a collection of resources where each resource represents a sensor data reading from a weather monitoring station. The following code shows the Django model for such station.

**Django Model for weather station – models.py:**

```
from django.db import models

class Station(models.Model):
    name = models.CharField(max_length=10)
    timestamp = models.CharField(max_length=10)
    temperature = models.CharField(max_length=5)
    lat = models.CharField(max_length=10)
    lon = models.CharField(max_length=10)
```

- Station model contains four fields – **station name, timestamp, temperature, latitude** and **magnitude.**

4. The following code shows the Django views for the REST API. ViewSets are used for the views that allow us to combine the logic for a set of related views in a single class.

❖Django View for weather station – views.py:

```
from myapp.models import Station
from rest_framework import viewsets
from django.shortcuts import render_to_response
from django.template import RequestContext
from myapp.serializers import StationSerializer
import requests
import json

class StationViewSet(viewsets.ModelViewSet):
    queryset = Station.objects.all()
    serializer_class = StationSerializer

def home(request):
    r=requests.get('http://127.0.0.1:8000/station/',
    auth=('username', 'password'))
    result=r.text
    output = json.loads(result)
    count=output['count']
    count=int(count)-1

    name=output['results'][count]['name']
    temperature=output['results'][count]['temperature']
    lat=output['results'][count]['lat']
    lon=output['results'][count]['lon']

    return render_to_response('index.html','name':name,
    'temperature':temperature, 'lat': lat, 'lon': lon,
    context_instance=RequestContext(request))
```

▪ViewDets are used for the views that allow one to combine the logic for a set of related views in a single class

5. The following code shows the serializers for the REST API. Serializers allow complex data (such as querysets and model instances) to be converted to native Python datatypes that can then be easily rendered into JSON, XML or other content types. Serializers also provide deserialization, allowing parsed data to be converted back into complex types, after first validating the incoming data.

**Serializers for Weather Station REST API - serializers.py**

```
from myapp.models import Station
from rest_framework import serializers

class StationSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Station
        fields = ('url', 'name','timestamp','timestamp',
        'temperature','lat','lon')
```

6. The following code shows the URL patterns for the REST API. As ViewSet is used instead of views, one can automatically generate the URL conf for API, by simply registering the viewsets with a router class. Routers automatically determining how the URLs for an application should be mapped to the logic that deals with handling incoming requests.

**Django URL patterns example - urls.py**

```
from django.conf.urls import patterns, include, url
from django.contrib import admin
from rest_framework import routers
from myapp import views

admin.autodiscover()

router = routers.DefaultRouter()
router.register(r'station', views.StationViewSet)

urlpatterns = patterns('',
url(r'^', include(router.urls)),
url(r'^api-auth/', include('rest_framework.urls',
namespace='rest_framework')),
url(r'^admin/', include(admin.site.urls)),
url(r'^home/', 'myapp.views.home'),
)
```

7. The following code shows the settings for REST API Django project.

**Django project settings example - settings.py**

```
DATABASES = {
'default': {
'ENGINE': 'django.db.backends.mysql',
'NAME': 'weatherstation',
'USER': 'root',
'PASSWORD': 'password',
'HOST': '',
'PORT': '',
}
}
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES':
    ('rest_framework.permissions.IsAdminUser',),
    'PAGINATE_BY': 10
}

INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'myapp',
    'rest_framework',
)
```

8. After creating the station REST API source files, the next step are to setup the database and then run the Django development webserver as follows:

```
■ python manage.py syncdb
python manage.py runserver
```

9. The following code shows examples of interacting with the station REST API using CURL. The HTTP POST method is used to create a new resource, GET method is used to obtain information about a resource, PUT method is used to update a resource and DELETE method is used to delete a resource.

**Using the Station REST API - CURL examples**

```
#------POST Example-------
$ curl -i -H "Content-Type: application/json" -H
"Accept:application/json; indent=4" -X POST -d
'"name": "CityName", "timestamp": "1393926310",
"temperature": "28", "lat": "30.733315", "lon":
"76.779418"' -u arshdeep http://127.0.0.1:8000/station/
Enter host password for user 'arshdeep':
HTTP/1.0 201 CREATED
Date: Tue, 04 Mar 2014 11:21:29 GMT
Server: WSGIServer/0.1 Python/2.7.3
Vary: Accept, Cookie
Content-Type: application/json; indent=4
Location: http://127.0.0.1:8000/station/4/
Allow: GET, POST, HEAD, OPTIONS

  "url": "http://127.0.0.1:8000/station/4/",
  "name": "CityName",
  "timestamp": "1393926310",
  "temperature": "28",
  "lat": "30.733315",
  "lon": "76.779418"
```

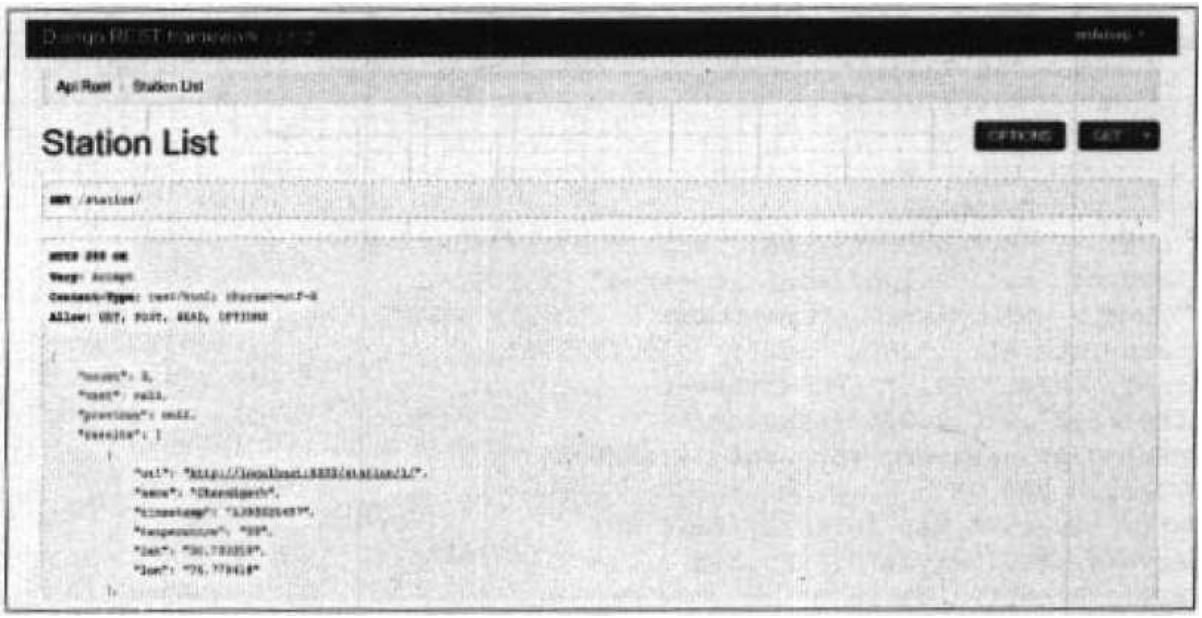
```
#------GET Examples-------
$ curl -i -H "Accept: application/json; indent=4"
http://127.0.0.1:8000/station/
Enter host password for user 'arshdeep':
HTTP/1.0 200 OK
Date: Tue, 04 Mar 2014 11:21:56 GMT
Server: WSGIServer/0.1 Python/2.7.3
Vary: Accept, Cookie
Content-Type: application/json; indent=4
Allow: GET, POST, HEAD, OPTIONS

{
    "count": 2,
    "next": null,
    "previous": null,
    "results": [
        {
            "url": "http://127.0.0.1:8000/station/1/",
            "name": "CityName",
            "timestamp": "1393926457",
            "temperature": "20",
            "lat": "30.733315",
            "lon": "76.779418"
        },
        {
            "url": "http://127.0.0.1:8000/station/2/",
            "name": "CityName",
            "timestamp": "1393926310",
            "temperature": "28",
            "lat": "30.733315",
            "lon": "76.779418"
        }
    ]
}
```

```
$ curl -i -H "Accept: application/json; indent=4" -u arsheep
http://127.0.0.1:8000/station/1/
Enter host password for user 'arshdeep':
HTTP/1.0 200 OK
Date: Tue, 04 Mar 2014 11:23:08 GMT
Server: WSGIServer/0.1 Python/2.7.3
Vary: Accept, Cookie
Content-Type: application/json; indent=4
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS

  "url": "http://127.0.0.1:8000/station/1/",
  "name": "CityName",
  "timestamp": "1393926457",
  "temperature": "20",
  "lat": "30.733315",
  "lon": "76.779418"
```

```
#-----PUT Example-------
$ curl -i -H "Content-Type: application/json" -H
"Accept: pplication/json; indent=4" -X PUT -d
'"name": "CityName", "timestamp": "1393926310",
"temperature": "29", "lat": "30.733315",
"lon": "76.779418"' -u arshdeep
http://127.0.0.1:8000/station/1/
Enter host password for user 'arshdeep':
HTTP/1.0 200 OK
Date: Tue, 04 Mar 2014 11:24:14 GMT
Server: WSGIServer/0.1 Python/2.7.3
Vary: Accept, Cookie
Content-Type: application/json
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS

url": "http://127.0.0.1:8000/station/1/",
name": "CityName", "timestamp": "1393926310",
temperature": "29", "lat": "30.733315", "lon": "76.779418"
```

```
#-----DELETE Example-------
$curl -i -X DELETE -H "Accept: application/json; indent=4" -u arshdeep
http://127.0.0.1:8000/station/2/
Enter host password for user 'arshdeep':
HTTP/1.0 204 NO CONTENT
Date: Tue, 04 Mar 2014 11:24:55 GMT
Server: WSGIServer/0.1 Python/2.7.3
Vary: Accept, Cookie
Content-Length: 0
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
```

10. The following figure shows the screenshot from web browsable station REST API.




## Amazon Web Services for IOT:

The following concepts explain how to use Amazon Web Services for IoT.

1. Amazon EC2
2. Amazon AutoScaling
3. Amazon S3
4. Amazon RDS
5. Amazon DynamoDB
6. Amazon Kinesis
7. Amazon SQS
8. Amazon EMR

**1. Amazon EC2:**

a. EC2 is an Infrastructure-as-a-Service (Iaas) provided by Amazon.

b. EC2 delivers scalable, pay as you go compute capacity in the cloud.

c. EC2 is a web service that provides computing capacity in the form of virtual machines that are launched in the Amazon's cloud computing environment.

d. EC2 can be used for several purposes of IoT.

**Ex:** IoT developers can deploy IoT applications on EC2, set up IoT platforms with REST API services.

The use of following Python code is to launch an instance. In this example, a connection to EC2 service is first established by calling **boto.ec2.connect_to_region**. The EC2 region, AWS access key and AWS secret key are passed to this function. After connecting to

EC2, a new instance is launched using the **conn.run_instances** function. The AMI_ID instance type, EC2 key handle and security group are passed to this function. This function returns a reservation. The instances associated with the reservation are obtained using **reservation.instances**. Finally the status of an instance associated with a reservation is obtained using the **instance.update** function. The program waits the status of the newly launched instance becomes **running** and then prints the instance details such as public DNS, instance IP and launch time.

**Python program for launching an EC2 instance**

```
import boto.ec2
from time import sleep

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"

REGION="us-east-1"
AMI_ID = "ami-d0f89fb9"
EC2_KEY_HANDLE = "<enter key handle>"
INSTANCE_TYPE="t1.micro"
SECGROUP_HANDLE="default"

print "Connecting to EC2"

conn = boto.ec2.connect_to_region(REGION,
   aws_access_key_id=ACCESS_KEY,
   aws_secret_access_key=SECRET_KEY)

print "Launching instance with AMI-ID %s, with keypair
%s, instance type %s, security group
%s"%(AMI_ID,EC2_KEY_HANDLE,INSTANCE_TYPE,SECGROUP_HANDLE)

reservation = conn.run_instances(image_id=AMI_ID,
        key_name=EC2_KEY_HANDLE,
        instance_type=INSTANCE_TYPE,
        security_groups = [ SECGROUP_HANDLE, ] )

instance = reservation.instances[0]

print "Waiting for instance to be up and running"

status = instance.update()
while status == 'pending':
   sleep(10)
   status = instance.update()

if status == 'running':
   print " \n Instance is now running.  Instance details are:"
   print "Intance Size:  " + str(instance.instance_type)
   print "Intance State:  " + str(instance.state)
   print "Intance Launch Time:  " + str(instance.launch_time)
   print "Intance Public DNS: " + str(instance.public_dns_name)
   print "Intance Private DNS: " + str(instance.private_dns_name)
   print "Intance IP: " + str(instance.ip_address)
   print "Intance Private IP: " + str(instance.private_ip_address)
```

The use of following Python code is used to stop an EC2 instance. In this example, the **conn.get_all_instances** function is called to get information on all running instances. This function returns reservations. Next, the IDs of instances associated with each reservation are obtained. The instances are stopped by calling **conn.stop_instances** function to which the IDs of the instances to stop are passed.

**Python program for stopping an EC2 instance**

```
import boto.ec2
from time import sleep
ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"

REGION="us-east-1"

print "Connecting to EC2"

conn = boto.ec2.connect_to_region(REGION,
   aws_access_key_id=ACCESS_KEY,
   aws_secret_access_key=SECRET_KEY)

print "Getting all running instances"
reservations = conn.get_all_instances()
print reservations

instance_rs = reservations[0].instances
instance = instance_rs[0]
instanceid=instance_rs[0].id
print "Stopping instance with ID: " + str(instanceid)

conn.stop_instances(instance_ids=[instanceid])

status = instance.update()
while not status == 'stopped':
   sleep(10)
   status = instance.update()

print "Stopped instance with ID: " + str(instanceid)
```

**2. Amazon AutoScaling:**

a. Amazon autoscaling allows automatically scaling Amazon EC2 capacity up or down according to user defined conditions.

b. With autoscaling the users can increase the number of EC2 instances running their applications seamlessly during their spikes in the application workloads to meet → application performance requirements.

c. scale down capacity when the workload is low to save costs.

d. Autoscaling can be used for autoscaling IoT applications and IoT platforms deployed on Amazon EC2.

The following Python code is used to create an AutoScaling group. In this example, a connection to AutoScaling service is first established by calling **boto.ec2.autoscale.connect_to_region** function.

**Python program for creating an AutoScaling group**

```
import boto.ec2.autoscale
from boto.ec2.autoscale import LaunchConfiguration
from boto.ec2.autoscale import AutoScalingGroup
from boto.ec2.cloudwatch import MetricAlarm
from boto.ec2.autoscale import ScalingPolicy
import boto.ec2.cloudwatch

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"

REGION="us-east-1"
AMI_ID = "ami-d0f89fb9"
EC2_KEY_HANDLE = "<enter key handle>"
INSTANCE_TYPE="t1.micro"
SECGROUP_HANDLE="default"

print "Connecting to Autoscaling Service"

conn = boto.ec2.autoscale.connect_to_region(REGION,
      aws_access_key_id=ACCESS_KEY,
      aws_secret_access_key=SECRET_KEY)


print "Creating launch configuration"


lc = LaunchConfiguration(name='My-Launch-Config-2',
        image_id=AMI_ID,
        key_name=EC2_KEY_HANDLE,
        instance_type=INSTANCE_TYPE,
        security_groups = [ SECGROUP_HANDLE, ])


conn.create_launch_configuration(lc)

print "Creating auto-scaling group"

ag = AutoScalingGroup(group_name='My-Group',
        availability_zones=['us-east-1b'],
        launch_config=lc, min_size=1, max_size=2,
        connection=conn)

conn.create_auto_scaling_group(ag)
```

```
print "Creating auto-scaling policies"

scale_up_policy = ScalingPolicy(name='scale_up',
            adjustment_type='ChangeInCapacity',
           as_name='My-Group',
            scaling_adjustment=1,
           cooldown=180)

scale_down_policy = ScalingPolicy(name='scale_down',
            adjustment_type='ChangeInCapacity',
           as_name='My-Group',
            scaling_adjustment=-1,
           cooldown=180)


conn.create_scaling_policy(scale_up_policy)
conn.create_scaling_policy(scale_down_policy)


scale_up_policy = conn.get_all_policies( as_group='My-Group',
policy_names=['scale_up'])[0]
scale_down_policy = conn.get_all_policies( as_group='My-Group',
policy_names=['scale_down'])[0]

print "Connecting to CloudWatch"

cloudwatch = boto.ec2.cloudwatch.connect_to_region(REGION,
        aws_access_key_id=ACCESS_KEY,
         aws_secret_access_key=SECRET_KEY)

alarm_dimensions = "AutoScalingGroupName":  'My-Group'

print "Creating scale-up alarm"

scale_up_alarm = MetricAlarm(
     name='scale_up_on_cpu', namespace='AWS/EC2',
     metric='CPUUtilization', statistic='Average',
     comparison='>', threshold='70',
     period='60', evaluation_periods=2,
     alarm_actions=[scale_up_policy.policy_arn] ,
     dimensions=alarm_dimensions)

cloudwatch.create_alarm(scale_up_alarm)

print "Creating scale-down alarm"

scale_down_alarm = MetricAlarm(
     name='scale_down_on_cpu', namespace='AWS/EC2',
     metric='CPUUtilization', statistic='Average',
     comparison='<', threshold='50',
     period='60', evaluation_periods=2,
     alarm_actions=[scale_down_policy.policy_arn],
     dimensions=alarm_dimensions)

cloudwatch.create_alarm(scale_down_alarm)
print "Done!"
```

**3. Amazon S3:**

a. Amazon S3 is an online cloud based data storage infrastructure for storing and retrieving a very large amount of data.
b. S3 provides highly reliable, scalable, fast, fully redundant and affordable storage infrastructure.
c. S3 can serve as a raw datastore for IoT systems for storing raw data, such as sensor data, image, audio and video data.
d. S3 Buckets are the fundamental container in Amazon S3 for data storage.
e. For others to access the objects in your S3 buckets, you'll need to explicitly grant them permissions

The use of following Python code is used to upload a file to Amazon S3 cloud storage. In this example, a connection to S3 service is established by calling **boto.connect_s3** function. This example defines two functions: **upload_to_s3_bucket_path** and **upload_to_s3_bucket_root**. The **upload_to_s3_bucket_path** function uploads the file to the S3 bucket specified at the specified path. The **upload_to_s3_bucket_root** function uploads the file to the S3 bucket root. (Page: 234)

**Python program for uploading a file to an S3 bucket**

```
import boto.s3

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"

conn = boto.connect_s3(aws_access_key_id=ACCESS_KEY,
   aws_secret_access_key=SECRET_KEY)

def percent_cb(complete, total):
   print ('.')

def upload_to_s3_bucket_path(bucketname, path, filename):
   mybucket = conn.get_bucket(bucketname)
   fullkeyname=os.path.join(path,filename)
   key = mybucket.new_key(fullkeyname)
   key.set_contents_from_filename(filename, cb=percent_cb, num_cb=10)

def upload_to_s3_bucket_root(bucketname, filename):
   mybucket = conn.get_bucket(bucketname)
   key = mybucket.new_key(filename)
   key.set_contents_from_filename(filename, cb=percent_cb, num_cb=10)

upload_to_s3_bucket_path('mybucket2013', 'data', 'file.txt')
```

**4. Amazon RDS:**

a. Amazon RDS is a web service that allows you to create instances of MySQL, Oracle or Microsoft SQL server in the cloud.
b. With RDS, developers can easily set up, operate and scale a relational database in the cloud.
c. RDS can serve as a scalable datastore for IoT systems.
d. With RDS, IoT system developers can store any amount of data in scalable relational databases.

The following Python code is used to launch an Amazon RDS instance. In this example, a connection to RDS service is first established by calling **boto.rds.connect_to_region** function. The RDS region, AWS access key and AWS secret key are passed to this function. After connecting to RDS service, the **conn.create_dbinstance** function is called to launch a new RDS instance. The input parameters to this function include the instance ID, database size, instance type, database user name & password, database port, database engine (e.g. MySql 5.1), database name, security groups etc. The program waits till the status of the RDS instance becomes available and then prints the instance details such as instance ID, create time, or instance end point.

**Python program for launching an RDS instance**

```
import boto.rds
from time import sleep

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"

REGION="us-east-1"
INSTANCE_TYPE="db.t1.micro"
ID = "MySQL-db-instance"
USERNAME = 'root'
PASSWORD = 'password'
DB_PORT = 3306
DB_SIZE = 5
DB_ENGINE = 'MySQL5.1'
DB_NAME = 'mytestdb'
SECGROUP_HANDLE="default"

print "Connecting to RDS"

conn = boto.rds.connect_to_region(REGION,
   aws_access_key_id=ACCESS_KEY,
   aws_secret_access_key=SECRET_KEY)

print "Creating an RDS instance"

db = conn.create_dbinstance(ID, DB_SIZE, INSTANCE_TYPE,
USERNAME, PASSWORD, port=DB_PORT, engine=DB_ENGINE,
db_name=DB_NAME, security_groups = [
SECGROUP_HANDLE, ] )
print db

print "Waiting for instance to be up and running"
```

```
status = db.status
while not status == 'available':
   sleep(10)
   status = db.status

if status == 'available':
   print " \n RDS Instance is now running.  Instance details are:"
   print "Intance ID: " + str(db.id)
   print "Intance State:  " + str(db.status)
   print "Intance Create Time:  " + str(db.create_time)
   print "Engine:  " + str(db.engine)
   print "Allocated Storage:  " + str(db.allocated_storage)
   print "Endpoint:  " + str(db.endpoint)
```

The following Python code can create a MySql table, writing and reading from the table. This example uses the MySqldb Paython package. To connect the MySqlRDS instance, the **MySQLdb.connect** function is called and the end point of the RDS instance, database username, password and port are passed to this function. After the connection to the RDS instance is established, a cursor to the database is obtained by calling **conn.cursor**. Next a new database table named **TemperatureData** is created with Id as primary key and other columns. After creating the table some values are inserted. To execute the SQL commands for database manipulation, the commands are passed to the **cursor.execute** function.

**Python program for creating a MySQL table, writing and reading from the table**

```
import MySQLdb

USERNAME = 'root'
PASSWORD = 'password'
DB_NAME = 'mytestdb'

print "Connecting to RDS instance"

conn = MySQLdb.connect (host =
"mysql-db-instance-3.c35qdifuf9ko.us-east-1.rds.amazonaws.com",
user = USERNAME,
passwd = PASSWORD,
db = DB_NAME,
port = 3306)

print "Connected to RDS instance"
```

```
cursor = conn.cursor ()
cursor.execute ("SELECT VERSION()")
row = cursor.fetchone ()
print "server version:", row[0]

cursor.execute ("CREATE TABLE TemperatureData(Id INT PRIMARY KEY,
Timestamp TEXT, Data TEXT) ")
cursor.execute ("INSERT INTO TemperatureData VALUES(1,
'1393926310', '20')")
cursor.execute ("INSERT INTO TemperatureData VALUES(2,
'1393926457', '25')")

cursor.execute("SELECT * FROM TemperatureData")
rows = cursor.fetchall()

for row in rows:
   print row

cursor.close ()
conn.close ()
```

**5. Amazon DynamoDB:**

a. Amazon DynamoDB is a fully managed, scalable, high performance "No_SQL" database service.

b. DynamoDB can serve as a scalable datastore for IoT systems.

c. With DynamoDB, IoT system developers can store any amount of data and serve any level of requests for the data.

The following Python code is used to create a DynamoDB table. In this example, a connection to DynamoDB service is first established by calling boto.dynamodb.connect_to_region. The DynamoDB region, AWS access key and AWS secret key are passed to this function. After connecting to DynamoDB service, a schema for the new table is created by calling conn.create_schema. The schema includes the hash key and range key names and types. A DynamoDB table is then created by calling conn.create_table function with the table schema, read units and write units as input parameters.

**Python program for creating a DynamoDB table**

```
import boto.dynamodb
import time
from datetime import date

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"
REGION="us-east-1"
```

```
print "Connecting to DynamoDB"

conn = boto.dynamodb.connect_to_region(REGION,
   aws_access_key_id=ACCESS_KEY,
   aws_secret_access_key=SECRET_KEY)

table_schema = conn.create_schema(
   hash_key_name='msgid',
   hash_key_proto_value=str,
   range_key_name='date',
   range_key_proto_value=str
   )

print "Creating table with schema:"
print table_schema


table = conn.create_table(
   name='my-test-table',
   schema=table_schema,
   read_units=1,
   write_units=1
   )

print "Creating table:"
print table

print "Done!"
```

The following Python code can write and read data from DynamoDB table. After establishing a connection with DynamoDB service, the **conn.get_table** is called to retrieve an existing table. The data written in this example consists of a JSON message with keys – **Body, CeatedBy and Time**. After creating the JSON message, a new DynamoDB table item is created by calling **table.new_item** and the hash key and range key is specified. The data item is finally committed to DynamoDB by calling **item.put**. To read data from DynamoDB, the **table.get_item** function is used with the hash key and range key as input parameters.

**Python program for writing and reading from a DynamoDB table**

```
import boto.dynamodb
import time
from datetime import date

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"
REGION="us-east-1"

print "Connecting to DynamoDB"

conn = boto.dynamodb.connect_to_region(REGION,
   aws_access_key_id=ACCESS_KEY,
   aws_secret_access_key=SECRET_KEY)
```

```
print "Listing available tables"
tables_list = conn.list_tables()
print tables_list

print "my-test-table description"
desc = conn.describe_table('my-test-table')
print desc

msg_datetime = time.asctime(time.localtime(time.time()))

print "Writing data"

table = conn.get_table("my-test-table")

hash_attribute = "Entry/" + str(date.today())

item_data =
   'Body':  'Test message',
   'CreatedBy':  'Vijay',
   'Time':  msg_datetime,


item = table.new_item(
   hash_key=hash_attribute,
   range_key=str(date.today()),
   attrs=item_data
)
item.put()
```

```
print "Reading data"

table = conn.get_table('my-test-table')

read_data = table.get_item(
      hash_key=hash_attribute,
      range_key=str(date.today())
      )

print read_data
print "Done!"
```

**6. Amazon Kinesis:**

a. Amazon Kinesis is a fully managed commercial service that allows real time processing of streaming data.

b. Kinesis scales automatically to handle high volume streaming data coming from large number of sources.

c. The streaming data collected by Kinesis can be processed by applications running on Amazon EC2 instances or any other compute instance that is connecting to Kinesis.

d. Kinesis is well suited for IoT systems that generate massive scale data and have strict real time. Requirements for processing the data.

e. Kinesis allows rapid and continuous data intake and support data blobs of size up to 50Kb.
f. The data producers write data records to Kinesis streams.
g. A data record comprises of a sequence number, a partition key and the data blob.
h. Data records in a Kinesis stream are distributed in shards.
i. Each shard provides a fixed unit of capacity and a stream can have multiple shards.
j. A single shard of throughput allows capturing 1MB per second of data, at up to 1000 PUT transactions per second and allows applications to read data up to 2 MB per second.

The following Python program can write the data to a Kinesis stream. This example follows a similar structure as the controller program that sends temperature data from an IoT device to the cloud. In this example a connection to the Kinesis service first established and then a new Kinesis stream is either created (if not existing) or described. The data is written to the Kinesis stream using the **kinesis.put_record** function.

**Python program for writing to a Kinesis stream**

```
import json
import time
import datetime
import boto.kinesis
from boto.kinesis.exceptions import ResourceNotFoundException
from random import randint

ACCESS_KEY = "<enter access key>"
SECRET_KEY = "<enter secret key>"

kinesis = boto.connect_kinesis(aws_access_key_id=ACCESS_KEY,
aws_secret_access_key=SECRET_KEY)
streamName = "temperature"
partitionKey = "IoTExample"
shardCount = 1
global stream


def readTempSensor():
#Return random value
   return randint(20,30)

#Controller main function
def runController():
   temperature = readTempSensor()
   timestamp = datetime.datetime.utcnow()
   record=str(timestamp)+":"+str(temperature)
   print "Putting record in stream:  " + record
   response = kinesis.put_record( stream_name=streamName,
data=record, partition_key=partitionKey)
   print ("-- put seqNum:", response['SequenceNumber'])
```

```
def get_or_create_stream(stream_name, shard_count):
   stream = None
   try:
      stream = kinesis.describe_stream(stream_name)
      print (json.dumps(stream, sort_keys=True, indent=2, separators=(',',
':  ')))
   except ResourceNotFoundException as rnfe:
      while (stream is None) or (stream['StreamStatus'] is not 'ACTIVE'):
         print ('Could not find ACTIVE stream:0 trying to create.'.format(
            stream_name))
         stream = kinesis.create_stream(stream_name, shard_count)
         time.sleep(0.5)

   return stream


def setupController():
   global stream
```

The following Python programs can the data from Kinesis stream. In this example, a shard iterator is obtained using the **kinesis.get_shard_iterator** function. The shard iterator specifies the position in the shard from which we want to start reading data records sequentially. The data is read using the **kinesis.get_records** function which returns one or more data records from a shard.

**Python program for reading from a Kinesis stream**

```
import json
import time
import boto.kinesis
from boto.kinesis.exceptions import ResourceNotFoundException
from boto.kinesis.exceptions import
ProvisionedThroughputExceededException
from boto.kinesis.exceptions import
ProvisionedThroughputExceededException

ACCESS_KEY = "<enter access key>"
SECRET_KEY = "<enter secret key>"

kinesis = boto.connect_kinesis(aws_access_key_id=ACCESS_KEY,
aws_secret_access_key=SECRET_KEY)
streamName = "temperature"
partitionKey = "IoTExample"
shardCount = 1
iterator_type='LATEST'

stream = kinesis.describe_stream(streamName)
print (json.dumps(stream, sort_keys=True, indent=2,
separators=(',', ':  ')))
shards = stream['StreamDescription']['Shards']
print ('# Shard Count:', len(shards))

def processRecords(records):
   for record in records:
      text = record['Data'].lower()
```

```
        print 'Processing record with data:  '  + text

i=0
response = kinesis.get_shard_iterator(streamName, shards[0]['ShardId'],
'TRIM_HORIZON', starting_sequence_number=None)
next_iterator = response['ShardIterator']
print ('Getting next records using iterator:  ', next_iterator)
while i<10:
   try:
      response = kinesis.get_records(next_iterator, limit=1)
      #print response
      if len(response['Records']) > 0:
         #print 'Number of records fetched:'  +
str(len(response['Records']))
         processRecords(response['Records'] )

      next_iterator = response['NextShardIterator']
      time.sleep(1)
      i=i+1

   except ProvisionedThroughputExceededException as ptee:
      print (ptee.message)
      time.sleep(5)
```

**7. Amazon SQS:**

a. Amazon SQS offers a highly scalable and reliable hosted queue for storing messages as they travel between distinct components of applications.
b. SQS guarantees only that messages arrive, not that they arrive in the same order in which they were put in the queue.
c. Amazon SQS may look similar to Amazon Kinesis; however both are intended for very different types of applications.
d. While Kinesis is meant for real time applications that involve high data ingress and egress rates, SQS is simply a queue system that stores and releases message in a scalable manner.
e. SQS can be used in distributed IoT applications in which various application components need to exchange messages.

The following Python can create an SQS queue. In this example, a connection to the SQS service is first established by calling **boto.sqs.connect_to_region**. The AWS region, access key and secret key are passed to this function. After connecting to SQS service, **conn.create_queue** is called to create a new queue with queue name as input parameter. The function **conn_get_all_queues** is used to retrieve all SQS queues.

**Python program for creating an SQS queue**

```
import boto.sqs

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"
REGION="us-east-1"

print "Connecting to SQS"

conn = boto.sqs.connect_to_region(
     REGION,
     aws_access_key_id=ACCESS_KEY,
     aws_secret_access_key=SECRET_KEY)

queue_name = 'mytestqueue'

print "Creating queue with name:  " + queue_name
q = conn.create_queue(queue_name)

print "Created queue with name:  " + queue_name

print " \n Getting all queues"

rs = conn.get_all_queues()

for item in rs:
   print item
```

The following Python code is for writing to an SQS queue. After connecting to an SQS queue, the **queue.write** is called with the message as input parameter.

**Python program for writing to an SQS queue**

```
import boto.sqs
from boto.sqs.message import Message
import time

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"

REGION="us-east-1"
```

```
print "Connecting to SQS"

conn = boto.sqs.connect_to_region(
     REGION,
     aws_access_key_id=ACCESS_KEY,
     aws_secret_access_key=SECRET_KEY)

queue_name = 'mytestqueue'

print "Connecting to queue:  " + queue_name
q = conn.get_all_queues(prefix=queue_name)
```

```
msg_datetime = time.asctime(time.localtime(time.time()))

msg = "Test message generated on:  " + msg_datetime
print "Writing to queue:  " + msg

m = Message()
m.set_body(msg)
status = q[0].write(m)

print "Message written to queue"

count = q[0].count()
```

The following Python code is for reading from an SQS queue. After connecting to an SQS queue, the queue.read is called to read a message from a queue.

**Python program for reading from an SQS queue**

```
import boto.sqs
from boto.sqs.message import Message

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"

REGION="us-east-1"

print "Connecting to SQS"

conn = boto.sqs.connect_to_region(
     REGION,
     aws_access_key_id=ACCESS_KEY,
     aws_secret_access_key=SECRET_KEY)

queue_name = 'mytestqueue'

print "Connecting to queue:  " + queue_name
q = conn.get_all_queues(prefix=queue_name)

count = q[0].count()

print "Total messages in queue:  " + str(count)

print "Reading message from queue"

for i in range(count):
   m = q[0].read()
   print "Message %d:  %s" % (i+1,str(m.get_body()))
   q[0].delete_message(m)

print "Read %d messages from queue" % (count)
```

**8. Amazon EMR:**

a. Amazon EMR is a web service that utilizes Hadoop framework running on Amazon EC2 and Amazon S3.

b. Amazon EMR allows processing of massive scale data, hence, suitable for IoT applications that generate large volumes of data that needs to be analyzed.
c. Data processing jobs are formulated with the MapReduce parallel data processing model.
d. MapReduce is a parallel data processing model for processing and analysis of massive scale data.
e. MapReduce model has two phases: Map and Reduce.
f. MapReduce programs are written in a functional programming style to create Map and Reduce functions.
g. The input data to the map and reduce phases is in the form of key-value pairs.

The following MapReduce example that finds the number of occurrences of a sequence from a log. The following Python code is written for launching an Elastic MapReduce job. In this example, a connection to EMR service is first established by calling **boto.emr.connect_to_region**. The AWS region, access key and secret key are passed to this function. After connecting to EMR service, a job flow step is created. There are two types of steps – streaming and custom jar. To create a streaming job an object of the **StreamingStep** class is created by specifying the job name, locations of the Mapper, Reducer, input and output. The job flow is then started using the **conn.run_jobflow** function with streaming step object as input. When the MapReduce job completes, the output can be obtained from the output location on the S3 bucket specified while creating the streaming step.

**Python program for launching an EMR job**

```
import boto.emr
from boto.emr.step import StreamingStep
from time import sleep

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"
REGION="us-east-1"

print "Connecting to EMR"

conn = boto.emr.connect_to_region(REGION,
      aws_access_key_id=ACCESS_KEY,
      aws_secret_access_key=SECRET_KEY)

print "Creating streaming step"

step = StreamingStep(name='Sequence Count',
      mapper='s3n://mybucket/seqCountMapper.py',
      reducer='s3n://mybucket/seqCountReducer.py',
      input='s3n://mybucket/data/',
      output='s3n://mybucket/seqcountoutput/')
```

```
print "Creating job flow"

jobid = conn.run_jobflow(name='Sequence Count Jobflow',
     log_uri='s3n://mybucket/wordcount_logs',
     steps=[step])

print "Submitted job flow"

print "Waiting for job flow to complete"

status = conn.describe_jobflow(jobid)
print status.state
```

```
while status.state != 'COMPLETED' or status.state != 'FAILED':
  sleep(10)
  status = conn.describe_jobflow(jobid)

print "Job status:  " + str(status.state)

print "Done!"
```

The following code shows the sequence count mapper program in Python. The mapper reads the data from standard input (stdin) and for each line in input in which the sequence occurs, the mapper emits a key-value pair where key is the sequence and value is equal to 1.

**Sequence count Mapper in Python**

```
#!/usr/bin/env python
import sys

#Enter the sequence to search
seq='123'
for line in sys.stdin:
  line = line.strip()
  if seq in line:
    print '%st%s' % (seq, 1)
```

The following code shows the sequence count reducer program in Python. The key-value pairs emitted by the map phase are shuffled to the reducers and grouped by the key. The reducer reads the key-value pairs grouped by the same key from the standard input (stdin) and sums up the occurrences to compute the count for each sequence.

**Sequence count Reducer in Python**

```
#!/usr/bin/env python
from operator import itemgetter
import sys

current_seq = None
current_count = 0
seq = None

for line in sys.stdin:
```

```
    line = line.strip()
    seq, count = line.split('t', 1)
    current_count += count

    try:
        count = int(count)
    except ValueError:
        continue

    if current_seq == seq:
        current_count += count
else:
    if current_seq:
        print '%st%s' % (current_seq,current_count)
    current_count = count
    current_seq = seq

if current_seq == seq:
    print '%st%s' % (current_seq, current_count)
```

## SkyNet IoT Messaging Platform:

a. SkyNet is an open source instant messaging platform for Internet of Things.

b. The SkyNet API supports both HTTP REST and real time WebSockets.

c. SkyNet allows users to register devices on the network.

d. A device can be anything including sensors, smart home devices, cloud resources, drones etc.

e. Each device is assigned a UUID and a secret token.

f. Devices or client applications can subscribe to other devices and receive/send messages.

The following commands are used to setup SkyNet on a Linux machine.

**Commands for Setting up SkyNet**

```
#Install DB Redis:
sudo apt-get install redis-server
#Install MongoDB:
sudo apt-key adv -keyserver keyserver.ubuntu.com -recv 7F0CEB10
echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart
dist 10gen' |
sudo tee /etc/apt/sources.list.d/10gen.list
sudo apt-get update
sudo apt-get install mongodb-10gen

#Install dependencies
sudo apt-get install git
sudo apt-get install software-properties-common
sudo apt-get install npm

#Install Node.JS
sudo apt-get update
sudo apt-get install -y python-software-properties python g++ make
sudo add-apt-repository ppa:chris-lea/node.js
sudo apt-get update
sudo apt-get install nodejs

#Install Skynet:
git clone https://github.com/skynetim/skynet.git
npm config set registry http://registry.npmjs.org/
npm install
```

The following code shows a sample configuration for SkyNet.

**Sample SkyNet configuration file**

```
module.exports = {
   databaseUrl:  "mongodb://localhost:27017/skynet",
   port:  3000,
   log:  true,
   rateLimit:  10, // 10 transactions per user per secend
   redisHost:  "127.0.0.1",
   redisPort:  "6379"
};
```

The following code shows examples of using SkyNet. The first step is to create a device on SkyNet. The POST request to create a device returns the UUID and token of the created device. This code also shows examples of updating a device, retrieving last 10 events related to a device, subscribing to a device and sending a message to a device.

**Using the SkyNet REST API**

```
#Creating a device
$curl -X POST -d "name=mydevicename&token=mytoken&color=green" or=green"
http://localhost:3000/devices
{"name":"mydevicename","token":"mytoken","ipAddress":"127.0.0.1",
"uuid":"myuuid","timestamp":1394181626324,"channel":"main",
"online":false,"_id":"531985fa16ac510d4c000006","eventCode":400}

---------------------
#Listing devices
$curl http://localhost:3000/devices/myuuid

{"name":"mydevicename","ipAddress":"127.0.0.1","uuid":"myuuid",
"timestamp":1394181626324, "channel":"main","online":false,
"_id":"531985fa16ac510d4c000006", "eventCode":500}

---------------------
#Update a device
$curl -X PUT -d "token=mytoken&color=red"
http://localhost:3000/devices/myuuid

#Get last 10 events for a device
$curl -X GET http://localhost:3000/events/myuuid?token=mytoken

{"events":[{"color":"red","fromUuid":"myuuid",
"timestamp":1394181722052, "eventCode":300,"id":"mytoken"},{
"name":"mydevicename","ipAddress":"127.0.0.1" ,
"uuid":"myuuid",
"timestamp":1394181626324, "channel":"main","online":false,
"eventCode":500, "id":"531985fa16ac510d4c000006"}]}

---------------------
#Subscribing to a device
$curl -X GET http://localhost:3000/subscribe/myuuid?token=mytoken

---------------------
#Sending a message
$curl -X POST -d '{"devices":  "myuuid", "message":
{"color":"red"}}' http://localhost:3000/messages
```

The following Python code shows for a client that performs various functions such as subscribing to a device, sending a message and retrieving the service status.

**Python client for SkyNet**

```
from socketIO_client import SocketIO

HOST='<enter host IP>'
PORT=3000
UUID='<enter UUID>'
TOKEN='<enter Token>'

def on_status_response(*args):
   print 'Status:  ', args

def on_ready_response(*args):
   print 'Ready:  ', args

def on_noready_response(*args):
   print 'Not Ready:  ', args

def on_sub_response(*args):
   print 'Subscribed:  ', args

def on_msg_response(*args):
   print 'Message Received:  ', args

def on_whoami_response(*args):
   print 'Who Am I : ', args

def on_id_response(*args):
   print 'Websocket connecting to Skynet with
socket id:  '  + args[0]['socketid']
   print 'Sending arguments:  ', type(args), args
   socketIO.emit('identity', 'uuid':UUID, 'socketid':
args[0]['socketid'], 'token':TOKEN)

def on_connect(*args):
   print 'Requesting websocket connection to Skynet'
   socketIO.on('identify', on_id_response)
   socketIO.on('ready', on_ready_response)
   socketIO.on('notReady', on_noready_response)
socketIO = SocketIO(HOST, PORT)
socketIO.on('connect', on_connect)

socketIO.emit('status', on_status_response)
socketIO.emit('subscribe', 'uuid':UUID,'token':  TOKEN, on_sub_response)

socketIO.emit('whoami', 'uuid':UUID, on_whoami_response)

socketIO.emit('message', 'devices':  UUID, 'message':  'color':'purple')
socketIO.on('message', on_msg_response)
socketIO.wait()
```

**********